conference

proceedings

The USENIX Windows NT Workshop Proceedings

Seattle, Washington
August 11–13, 1997

Sponsored by
The USENIX Association

USENIX®
The Advanced Computing Systems Association

USENIX Windows NT Workshop Proceedings
Seattle, Washington August 1997

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $18 for members and $24 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-88-X

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

USENIX

Windows NT Workshop

August 11-13, 1997
Seattle, Washington

Workshop Organizers

Program Co-Chairs

Michael B. Jones, *Microsoft Research*
Ed Lazowska, *University of Washington*

Steering Committee

Trevor Mudge, *University of Michigan*
Margo Seltzer, *Harvard University*
Thorsten von Eicken, *Cornell University*

Program Committee

Tom Anderson, *University of California, Berkeley*
Bryant Bigbee, *Intel Corporation*
Bill Carpenter, *VenturCom, Inc.*
J. Bradley Chen, *Harvard University*
Anton Chernoff, *Digital Equipment Corporation*
Andrew A. Chien, *University of Illinois at Urbana-Champaign*
Jim Gray, *Microsoft Bay Area Research Center*
Carl Hauser, *Xerox Palo Alto Research Center*
Avi Mendelson, *Technion*
Trevor Mudge, *University of Michigan*
Richard Oehler, *IBM T.J. Watson Research Center*
Thorsten von Eicken, *Cornell University*

External Reviewers

Peter Bird
Bruce Jacob
Karin Petersen

Special Session Organizers

J. Bradley Chen—*Tuturial Session: Available Tools*
Thorsten von Eicken—*Panel Session: Do You Need Source?*
Ed Lazowska and Jim Gray—*Invited Talks and Panel: Building Distributed Applications: CORBA and DCOM*
Avi Mendelson—*Demonstrations and Posters*
Jim Gray and Richard Oehler—*Case Studies: Deep Ports*
Michael B. Jones—*Invited Talk: Windows NT Futures*

USENIX Association Staff

Judith F. DesHarnais, *Meeting Planner*
Ellie Young, *Executive Director*
Eileen Cohen, *Publications Director*
Zanna Knight, *Marketing Director*

Table of Contents

USENIX Windows NT Workshop

August 11-13, 1997
Seattle, Washington

Monday, August 11

Tuesday, August 12

Wednesday, August 14

Message from the Program Chairs

Welcome to the USENIX Windows NT Workshop!

Increasing numbers of researchers are using or adopting Windows NT as their research base. The USENIX Windows NT Workshop is intended to provide a forum for these researchers to discuss ideas and share information, experiences, and results.

But even beyond these normal workshop functions, we have another goal for the workshop: community-building among researchers using Windows NT. Researchers using UNIX benefit from a 20+ year history of meeting together, trading experiences, expertise, research results, scuttlebutt, and war stories. (And of course, the USENIX Association has played a major role in making this happen.) If the workshop helps bootstrap a similar community of Windows NT users, we will have accomplished our real purpose.

We'd like to thank the 36 authors who took a chance on an unproven workshop. We believe you'll agree with the committee that the quality of submissions was quite high, resulting in a very interesting program. 17 of these submissions were selected for presentation during the refereed paper sessions, and several more will be presented during the posters and demonstrations session.

We'd like to thank the program committee for helping put together an excellent program. The committee members are Tom Anderson, Bryant Bigbee, Bill Carpenter, Brad Chen, Anton Chernoff, Andrew Chien, Jim Gray, Carl Hauser, Avi Mendelson, Trevor Mudge, Richard Oehler, and Thorsten von Eicken. Our special thanks go to Jim Gray for his service way above and beyond the call of duty.

Other people key to making the workshop happen were Peter Honeyman, who first talked us into organizing it, and Margo Seltzer, who has helped behind the scenes in numerous ways. Finally, we want to state for the record that the USENIX staff has been a joy to work with. In particular, Eileen Cohen, Judy DesHarnais, Zanna Knight, Toni Veglia, and Ellie Young all deserve credit. They've made sure that the myriad details of organizing a workshop just happened, freeing us to focus on the content.

It's been fun. We're looking forward to interacting with many of the you at the workshop!

Michael B. Jones
Ed Lazowska

August, 1997

Instrumentation and Optimization of Win32/Intel Executables Using Etch

Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman,
Wayne Wong, Hank Levy, and Brian Bershad
University of Washington

Brad Chen
Harvard University

etch-info@cs.washington.edu

Abstract

Etch is a general-purpose tool for rewriting arbitrary Win32/x86 binaries without requiring source code. Etch provides a framework for modifying executables for both measurement and optimization. Etch handles the complexities of the Win32 executable file format and the x86 instruction set, allowing tool builders to focus on specifying transformations. Etch also handles the complexities of the Win32 execution environment, allowing tool users to focus on performing experiments. This paper describes Etch and some of the tools that we have built using Etch, including a hierarchical call graph profiler and an instruction layout optimization tool.

1 Introduction

During the last decade, the Intel x86 instruction set has become a mainstay of the computing industry. Arguably, Intel processors have executed more instructions than all other computers ever built. Despite the widespread use of Intel processors and applications, however, few tools are available to assist the programmer and user in understanding or exploiting the interaction between applications, the processor, and the memory system on x86-based platforms. At the University of Washington, we have been building a software architecture based on *binary rewriting* for developing such tools on Intel x86 platforms running Win32.

This paper describes **Etch**, a binary rewriting system for Win32 applications that run on Intel x86 processors. We developed Etch to aid programmers, users, and researchers in understanding the behavior of arbitrary applications running on Intel architectures. Application source is not required to rewrite a program. Etch supports general transformations on programs, so that they can also be rewritten to optimize their performance, *without* access to the original source code.

Etch is targeted at three different user groups: *architects*, who wish to understand how current application workloads interact with the architecture of a computer system; *developers*, who wish to understand the performance of their programs during the development cycle; and *users*, who wish to understand and improve the performance of common applications executing in their environment. Etch provides all three groups with measurement tools to evaluate performance at several levels of detail, and optimization tools to automatically restructure programs to improve performance, where possible.

Previous binary modification tools such as pixie [Chow et al. 1986], ATOM [Srivastava & Eustace 1994] and EEL [Larus & Schnarr 1995] run on RISC-based UNIX systems. Other tools for modifying x86 programs, such as MPTrace [Eggers et al. 1990], run on UNIX systems and require access to compiler-generated assembly language versions of the input programs. In contrast, Etch, like TracePoint's *Hiprof* call graph profiling tool [TracePoint 1997], works directly on Win32 x86 binaries. This environment creates several challenges that are not present in UNIX-based environments, including:

- *code discovery:* The complexity of the x86 instruction set and the current practice of interleaving data and executable instructions in the text segment make it difficult to statically discover code within an executable image. Although the structure of the Win32 PE header is well defined [Pietrek 1994], there is no standard that defines an executable's internal structure. Internally, a binary can contain code, data, and jump tables in an arbitrary order, and the format commonly changes from compiler to compiler. A binary rewriting tool must be able to accurately distinguish between code and data so that it can rewrite code while leaving data intact. Failure to make this distinction can result in a broken executable or in large amounts of uninstrumented code.
- *module discovery:* Win32 applications are commonly composed of a top level executable and a large number[1] of dynamically linked libraries (DLLs). Al-

[1] As an example, Lotus Wordpro 96 loads 41 DLLs during a simple use of the application, of which 17 are listed in the executable header and 24 are identified during the run of the application.

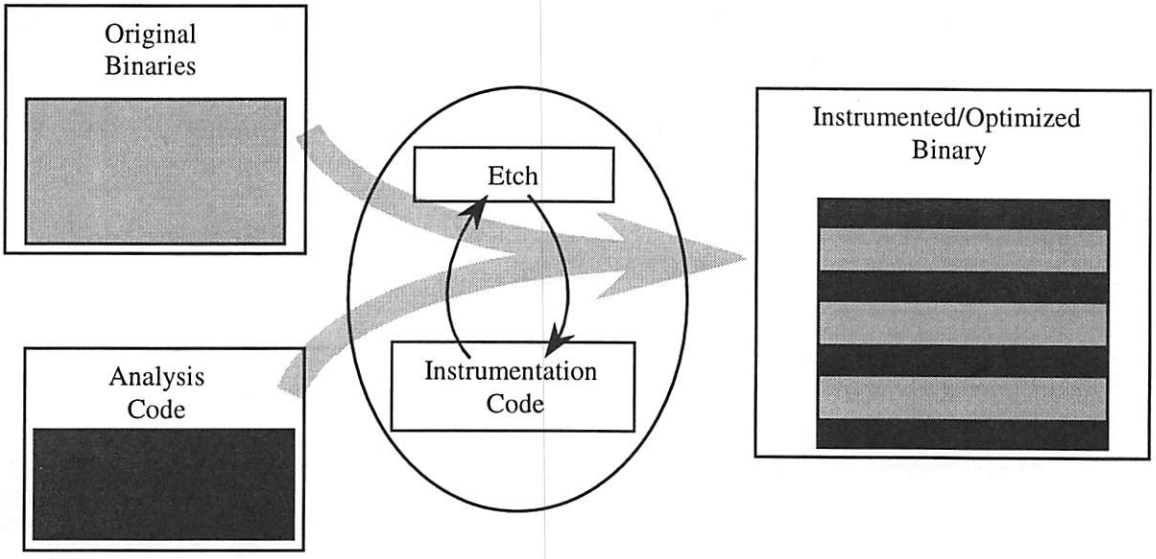


Figure 1: Etch transforms a program according to some instrumentation code, which can add new code into the program, take old code out, or simply reorder pieces of the program.

though some of the libraries may be statically identified in the executable headers, many are identified and loaded dynamically. In this case the names of the DLLs to be loaded by an application can only be determined by running the program. Transformation of such a loosely structured application requires that an instrumentation tool be able to identify all the DLLs used by a program. In contrast, most UNIX applications are composed of a single statically loaded executable, or an executable and a relatively small number of shared libraries.

- *environment management:* A Win32 application executes in a context, which includes its working directory and the program name by which it was invoked. Unlike typical UNIX applications, many Win32 applications are sensitive to this context and may not work when the module used by the application is copied or moved into a different directory. Consequently, it is necessary to run a transformed executable within a protective "shell" that makes it appear as though the Win32 executable is running in its original context.

2 The Model

Etch is a system framework that supports integration of nearly any type of measurement and optimization tool, similar to systems such as ATOM and EEL. Etch permits measurement tools to instrument an x86 binary so that the application program, when executed, produces the required measurement data as it runs. For example, one tool may instrument the program to produce a trace of all memory reads and writes, while another may instrument the program to record the number of conditional branch instructions that succeed or fail. Furthermore, because Etch allows complex modifications to the executable binary, Etch tools can also rewrite the program in order to improve its performance. For example, an Etch tool may reorder instructions to optimize for the pipeline structure of a particular processor implementation, or it may relocate procedures to improve memory system locality and behavior.

Etch separates the process of instrumenting and tracing an executable into two phases: an instrumentation phase and an analysis phase. Similarly, each tool is split into two components, an instrumentation module and an analysis module. During the instrumentation phase, Etch processes the program in order to "discover" the components of the program, e.g., instructions, basic blocks, and procedures. As Etch discovers each component, it calls into the tool's instrumentation module, telling the module what it discovered. At each such callback, the instrumentation module has the opportunity to instruct Etch to examine and possibly modify the executable, e.g., to insert measurement instructions before or after an instruction, basic block, or procedure. These inserted instructions may include calls to procedures in the analysis module, which will be loaded with the executable at run time (the analysis phase). Thus, when the program runs, it will execute the additional inserted code, including calls into the tool analysis routines that record or process crucial measurement information. Finally, when the program completes, the analysis module is given an opportunity to run analysis routines required to process the data collected during execution.

```
InstrumentModule(thisModule, Before)
For each procedure in the program:
    InstrumentProcedure(thisProcedure, Before)
    For each basic block in the Procedure:
        InstrumentBasicBlock(thisBasicBlock, Before)
        For each instruction in the Basic Block:
            InstrumentInstruction(thisInstruction, Before)
            InstrumentInstruction(thisInstruction, After)
        InstrumentBasicBlock(thisBasicBlock, After)
    InstrumentProcedure(thisProcedure, After)
InstrumentModule(thisModule, After)
```

Figure 2: As Etch discovers program components during program instrumentation, it invokes instrumentation code for that component: once before the component is written to the new executable, and once after. The implementations of the Instrument routines may direct Etch to add new code before and/or after the specified*

Figure 1 illustrates the general transformation of an executable using Etch. As Etch discovers pieces of the original executable, it invokes the instrumentation code in the manner indicated in Figure 2. The instrumentation tool provides implementations of "Before" and "After" functions. These functions can in turn direct Etch to modify the executable with respect to the specific instruction. The directions in effect say "before (or after) this instruction runs, please call some specific function with some specific set of arguments." For example, to count instructions, the InstrumentBefore procedure would direct Etch to insert a call to a procedure that incremented a counter at run time. Instrumentation code can add, remove, or modify instructions, or add procedure calls at any point in the executable. There are provisions for communicating program state (such as register values) into analysis code at run time. By default, all program state is with respect to the original executable.

Once the entire executable has been traversed, Etch generates a new version of the executable that includes the instructions added during instrumentation. The functions called at instrumentation points, as well as the Etch run time library, are linked (dynamically loaded) by the new executable. When the executable written by Etch is run, analysis routines will run as a side effect of running the program. These instrumentation routines can inspect the state of the program, for example, the contents of registers, or effective addresses. All addresses, whether text or data, are relative to the original binary, so the collection routines do not have to reverse-map these addresses at run time.

The transformations performed on the binary by Etch should not change program correctness, although it

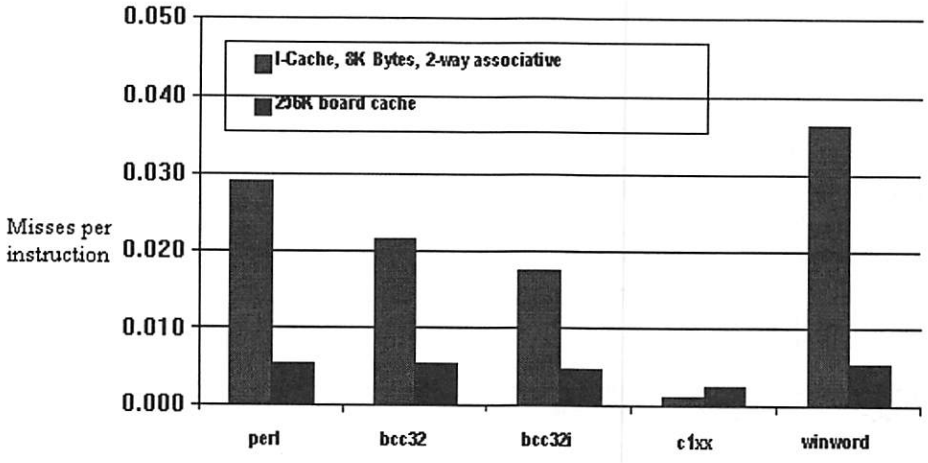


Figure 3: Cache miss data for a collection of popular NT applications.

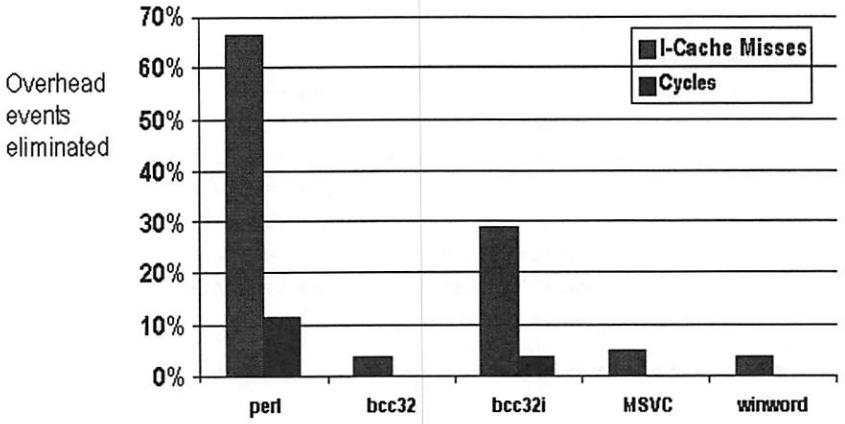


Figure 4: The performance impact of code layout optimization.

is of course possible to write instrumentation or analysis code that causes a program to break when it executes. A program transformed for performance measurement collection may run more slowly depending on the number of additional instructions that must be executed. Etch does not require changes to the operating system, but Etched binaries may utilize OS facilities, such as software timers, or even implementation-specific facilities, such as Intel Pentium performance counters.

2.1 Measurement

A typical Etch measurement tool analyzes an application's behavior as it runs, and at program termination saves information about the run to disk. For example, a cache simulation tool could examine the application's memory reference stream and report cache miss rates for various cache configurations. A post-processing tool could then predict the application's execution time based on the cache miss rates and hypothetical processor, cache, and memory speeds. A different tool could simply display the cache miss data graphically, as illustrated in Figure 3. The graph shows the number of misses per instruction in the first level instruction cache and the second level unified cache for the Perl interpreter, three commercial C++ compilers, and Microsoft Word.

2.2 Optimization

Etch also provides facilities for rewriting an executable in order to improve its performance. For example, it is possible to reorder instructions to optimize code layout for cache and VM behavior.

Figure 4 shows the reduction in instruction cache misses and execution time (in cycles) for a collection of

```
Call Graph Profile:
    parents
name                                          %time  %self  %desc  calls
    children

"SYNCH.EXE",_threadstart [1:29]                89.0    0.0   89.0  2
    "SYNCH.EXE",CopyToScreen [1:1]                     0.0   88.0  2
    DllCall:,"KERNEL32.dll",TlsSetValue [0:12]         0.0    0.0  2
    "SYNCH.EXE",_endthread [1:30]                      0.0    0.0  2
-------------------------------------------------
    "SYNCH.EXE",_threadstart [1:29]                    0.0   88.0  2
"SYNCH.EXE",CopyToScreen [1:1]                 89.0    0.0   88.0  2
    "SYNCH.EXE",Lock::Release [1:9]                    0.0    0.0  4
    "SYNCH.EXE",Lock::Acquire [1:8]                    0.0    0.0  4
    "SYNCH.EXE",getc [1:20]                            0.0    4.0  1310
    "SYNCH.EXE",Condition::Signal [1:13]               0.0    0.0  2
    "SYNCH.EXE",fopen [1:27]                           0.0    1.0  2
    "SYNCH.EXE",putchar [1:22]                         0.0   80.0  1308
```

Figure 5: Excerpt of typical hierarchical call graph profiler output. For each procedure the output includes: the name of the module (executable or DLL); the name of the procedure itself; a unique identifier for the procedure; the time spent in the procedure, in its callers (parents) and in its callees (children); and the number of calls from each caller to the procedure and from the procedure to each callee.

popular Win32 programs that have been optimized for code layout using Etch on a 90Mhz Pentium. Etch was first used to discover the programs' locality while executing against a training input, and then to rewrite the applications, in order to achieve a tighter cache and VM packing. Infrequently executed basic blocks were moved out of line, and frequently interacting blocks were laid out contiguously in the executable, using an algorithm based on Pettis and Hansen [Pettis & Hansen 1990]. The results were measured using hardware performance counters. Different inputs were used for training and testing.

2.3 Call Graph Profiling

An example of a relatively complex Etch tool is the Etch call-graph profiler, CGProf. CGProf shows program activity in terms of the program's dynamic call-graph, a format originally used by the UNIX gprof profiler [Graham et al. 1983]. CGProf is designed to give precise and complete information about the bottlenecks and time-sinks in an application. Etch and CGProf can use debugging information to provide output using procedure names. When source code is also available, a coordinated GNU Emacs browsing mode provides point-and-click navigation of source from CGProf profiles. CGProf provides multiple views of profile information to help developers identify bottlenecks at different structural levels. These views include a per-module and per-procedure view, to identify the modules and procedures where execution time is spent, as well as the hierarchical view illustrated in Figure 5.

CGProf uses Etch instrumentation to monitor all procedure calls and returns during execution rather than statistical sampling commonly found in other profilers. CGProf can count either cycles or instructions to quantify activity in application modules. When counting cycles, the tool uses the hardware cycle counter to measure path lengths. When counting instructions, CGProf instruments every basic block to add the length of the basic block to a counter.

3 Using Tools Written With Etch

There are three basic ways to use tools developed with Etch: one-at-a-time by hand; using a GUI; or through a tool-specific wrapper from the command line. The first, and most primitive, is to simply invoke Etch for each component in a program, passing in as an argument the name of the tool and the program component. For complex programs, this can be somewhat tedious and error prone, as it is necessary to specify each program component one at a time.

The second and more convenient way to run Etch is to use a graphical user interface. The Etch GUI (Visual Etch) makes a standard collection of tools available to Etch users without requiring that they write or understand the mechanics of building Etch tools. Our goal in

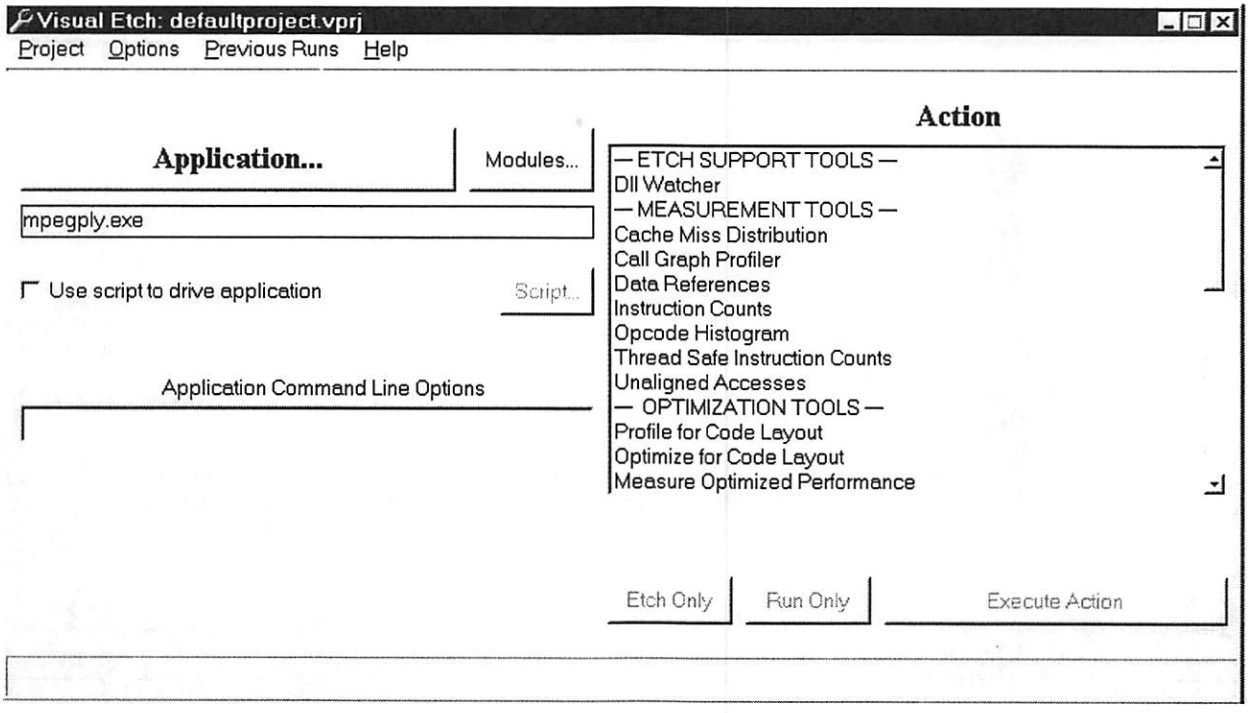


Figure 6: The Visual Etch user interface.

developing the GUI was to make it extremely easy for the naïve user (such as an undergraduate programmer in an architecture course) to be able to run sophisticated experiments on commercial software. As such, it very much follows the "point, click, and go" model of control as illustrated by Figure 6. The user simply specifies a program and a tool, runs the program as modified by the tool, and then looks at the output.

The user interface determines all of the program components, drives the rewriting process and ensures that whatever environment (e.g., working directory, data files, etc.) was initially available to the original executable is available to the transformed executable at run time. The GUI first runs Etch on the original binary to produce a new binary that has been modified to collect the necessary data. It executes the modified binary to produce the data, and feeds the data to analysis tools that produce graphs or charts that illustrate behavior or pinpoint problems. For example, Figure 7 shows the output from an opcode histogram tool that displays the distribution of instruction types for an MPEG player. If Etch is being used to optimize performance, the user may instruct Etch to apply a performance-optimization transformation. For example, Etch may rewrite the original binary to change the layout of data or code in order to improve cache or virtual memory performance, as was illustrated in Figure 4.

The third way to run Etch is by using a command line version of the instrumentation front end. Our command line front end is based on a generic wrapper program that is specialized at compile time to the specific instrumentation tool with which it will be used. For example, we compile the wrapper along with the CGProf libraries to build the program CGInstrument. CGInstrument can then be used to instrument applications for call-graph profiling. Once instrumented, the profiled application can be run, and then another command, CGProfile , is used to post-process the raw profile information. For example, the sequence of commands to instrument and profile notepad.exe would be:

```
C:> cginstrument notepad.exe
C:> notepad-cgprof.exe
C:> cgprofile notepad.exe
```

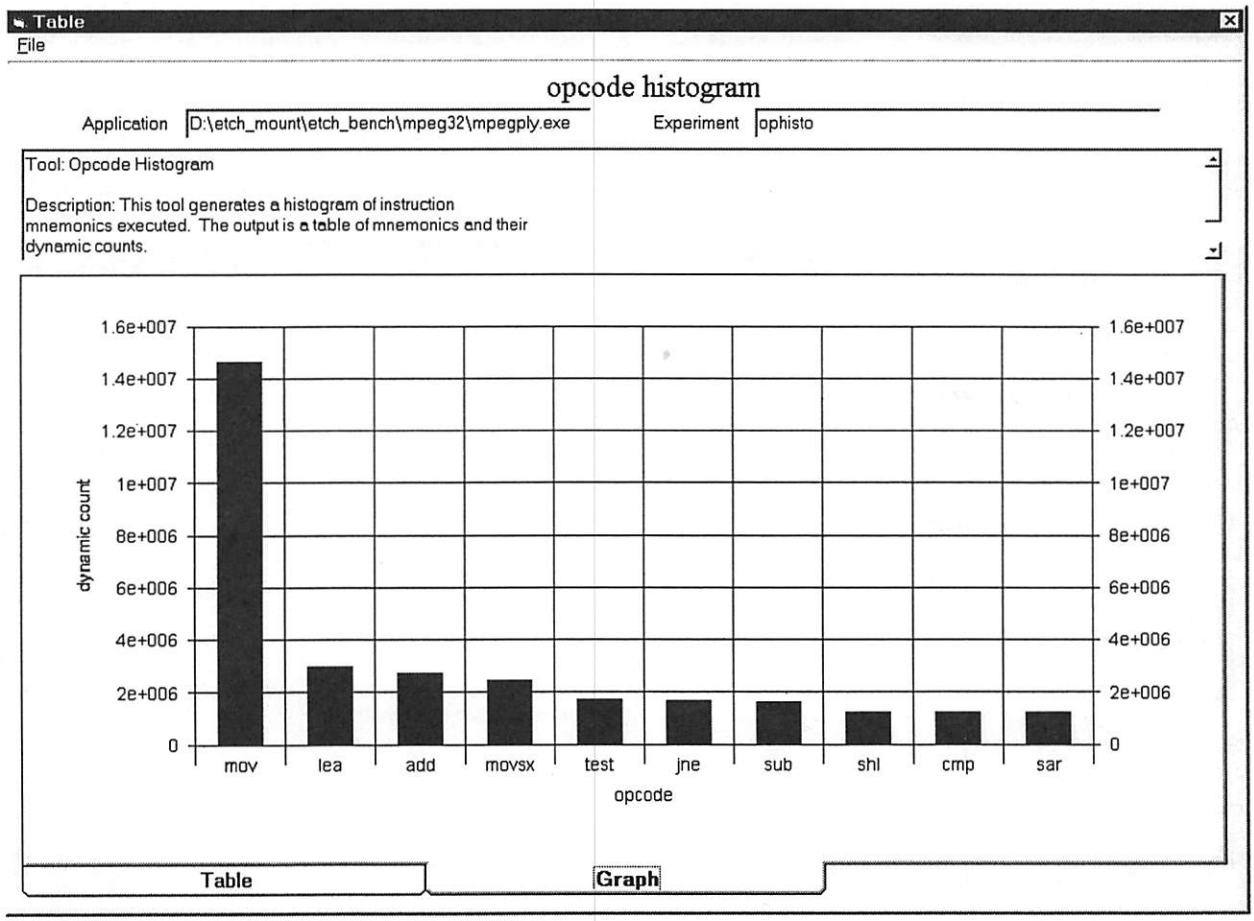


Figure 7: Sample results showing distribution of instruction opcodes.

The first step takes the original program (in this case notepad.exe) and all of its DLLs and produces versions that have been transformed to collect profile information. The second step runs the transformed version of the program (and uses all of the transformed DLLs), calling into the run time analysis routines to generate an output file containing profile information. Finally, the third step converts this output file into human-readable text.

The CGProf command line interface makes it possible to include profiling with CGProf as a part of an automated build/test environment, rather than requiring interaction with a GUI. The CGProf command line interface manages the discovery and instrumentation of all the modules used by an application. It can watch the application during a training run to build the entire list of modules used by the application for a specific test and make the list available to the instrumentation process. Alternatively, new modules can be detected and instrumented while the profiling experiment runs. This makes it possible to eliminate the training run, and to accommodate slight variations in testing runs that cause new DLLs to be loaded.

4 Summary

Etch is a general binary rewriting tool for Win32 executables running on Intel architectures. Its key features are a generalized API that allows tools to be developed relatively quickly and run with relatively good performance. To learn more about the Etch project, or to obtain a version of this paper with color figures, please visit:

http://www.cs.washington.edu/homes/bershad/etch

For traces generated from a few popular Win32 programs on the x86, visit:

http://etch.eecs.harvard.edu/traces/index.html

References

[Chow et al. 1986]
Fred Chow, A. M. Himelstein, Earl Killian and L. Weber, `"Engineering a RISC Compiler System," *IEEE COMPCON*, March 1986.

[Eggers et al. 1990]
S. Eggers, D. Keppel, E. Koldinger and H. Levy, "Techniques for Efficient Inline Tracing on a Shared Memory Multiprocessor," *Proceedings of the ACM Conference on Measurement and Modeling of Systems,* May 1990.

[Graham et al. 1983]
S. Graham, P. Kessler, and M. McKusick. "An Execution Profiler for Modular Programs," *Software - Practice and Experience*, 13, pp. 671-685, 1983.

[Larus & Schnarr 1995]
James R. Larus and Eric Schnarr, "EEL: Machine-Independent Executable Editing", *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Impelementation (PLDI),* June 1995.

[Pettis & Hansen 1990]
Karl Pettis and Robert Hansen, "Profile-Guided Code Positioning", *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* June 1990.

[Pietrek 1994]
Matt Pietrek, "Peering Inside PE: A Tour of the Win32 Portable Executable Format", *Microsoft Systems Journal,* Vol. 9, No. 3, pg 15-34, March 1994.

[Srivastava & Eustace 1994]
A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* June 1994. See also DEC WRL Research Report 94/2.

[TracePoint 97]
http://tracepoint.galatia.com/frames.html

DIGITAL FX!32
Running 32-Bit x86 Applications on Alpha NT

Anton Chernoff, Ray Hookway
Digital Equipment Corporation

Abstract

DIGITAL FX!32 is a unique combination of emulation and binary translation which makes it so that any 32-bit program that runs on an x86 system running Windows NT 4.0 will install and run on an Alpha Windows NT 4.0 system. After translation, x86 applications run as fast under DIGITAL FX!32 on a 500Mz Alpha system as on a 200Mz Pentium-Pro.

The emulator and its associated runtime provide for transparent execution of x86 applications. The emulator uses translation results when they are available and produces profile data for use by the translator. The translator provides native Alpha code for the portions of an x86 application which have been previously executed. A server manages the translation process for the user, making the overall process completely transparent.

This paper focuses on the ways in which DIGITAL FX!32 achieves its transparency when running on an unmodified NT system.

1. Introduction

Three factors contribute to the success of a microprocessor: price, performance, and software availability. DIGITAL FX!32 addresses the last of these factors, software availability, by making hundreds of new applications available on Alpha-based platforms running Windows NT. DIGITAL FX!32 combines emulation and binary translation to provide fast, transparent execution of x86 programs on Alpha.

Since it was introduced, the Digital Alpha microprocessor has been the fastest microprocessor available. A large number of applications, particularly those that require a high performance processor, are available on Alpha. DIGITAL FX!32, makes it so that any 32-bit program which runs on an x86 system running Windows NT 4.0 will install and run on an Alpha Windows NT 4.0 system. The performance of an x86 application running on a high end Alpha is similar to the performance of the same application running on a high end x86 platform.

Many different systems have successfully used emulators to run applications on platforms for which they were not initially targeted[4, 7]. The major drawback has been poor performance[7]. Several emulators have used "dynamic translation" to achieve better performance than that which a straight interpreter can obtain[2, 3, 7]. This approach translates small segments of the program while it is being executed. These systems must make a tradeoff between the amount of time spent translating and the resulting benefit of the translation. Too much time spent on the translation and related processing makes the program unresponsive. This limits the optimizations that emulators can perform using dynamic translation.

DIGITAL FX!32 makes a different tradeoff. No translation is done while the application is executing. Rather, the emulator captures an execution profile. Later, a binary translator[1] uses the profile to translate the parts of the application that have been executed into native Alpha code. Since the translator runs in the background, it can use computationally intensive algorithms to improve the quality of the generated code. To our knowledge, DIGITAL FX!32 is the first system to explore this mix of emulation and binary translation.

One of the important features of DIGITAL FX!32 is the transparent execution of x86 programs. Digital has provided several static binary translators. These were targeted at developers and sophisticated end-users. They required the user to manually use the translation tool to convert code from one computer architecture to another. This scheme was difficult to use when confronted with a distribution kit containing many images. With the ascension of the point-and-click user interface, a static translator is not feasible -- users expect programs to "just work." DIGITAL FX!32 achieves this transparency in a number of ways. Other than specifying that an application is an x86 application when it is being installed, the process of installing and running the application is the same on an Alpha as it is on an x86.

2. Overview

Windows NT has used an emulator to run 16-bit x86 applications since it was first released. Applications which run on this emulator install and run just like they do on an x86, but they run much slower. The emulator built into DIGITAL FX!32 provides a similar capability for 32-bit applications.

Unlike the 16-bit environment, DIGITAL FX!32 also provides a binary translator which translates 32-bit x86 applications into native Alpha code. The translation is done in the background and requires no interaction with the user. Operating in the background allows the DIGITAL FX!32 translator to perform optimizations that are too expensive to perform while the application is running. The resulting translated application runs up to ten times faster than the same application running under the emulator.

DIGITAL FX!32 consists of seven major components. Along with the emulator and translator mentioned above, the other major components are the agent, the runtime, the database, the server and the manager. The agent provides for transparent launching of 32-bit x86 applications. The runtime loads x86 images and sets up the runtime environment to execute them. As part of loading an image, the runtime "jackets" imported API routines. These jackets allow the x86 code to call the native Windows API. The database stores execution profiles produced by the emulator and used by the translator. It also stores translated images. The server maintains the database and runs the translator as appropriate. The manager allows the user to control resources used by DIGITAL FX!32. Each of the major components is discussed in more detail below.

This paper will focus on the agent and the runtime, since those components are primarily responsible for achieving the run-time transparency.

3. The DIGITAL FX!32 Agent

The DIGITAL FX!32 agent provides for transparent launching of 32-bit x86 applications. It is a DLL that is inserted into the address space of a process and hooks calls on CreateProcess and related APIs. If a call to CreateProcess specifies an x86 image to be executed, the Agent invokes the DIGITAL FX!32 runtime to execute the image instead. A process which contains the Agent is said to be *enabled*.

3.1 Enabling a Process

DIGITAL FX!32 enables processes using a technique described by Jeffrey Richter in chapter 16 of his book Advanced Windows NT[8] to inject a copy of the agent into the process' address space.

The process doing the enabling (the *enabler*) must have a handle on the process being enabled (the *subject*). For a process created by the enabler, this is the value returned by the CreateProcess call. Other enablers must get a handle with OpenProcess. This frequently requires administrator privileges.

The enabler allocates a small area of virtual memory in the address space of the subject by starting a suspended thread (CreateRemoteThread) and using its stack. It changes the protection of that memory to executable, readable, and writable (WriteProcessMemory). It then copies a small piece of code and data into the subject (WriteProcessMemory). The code that it copies simply calls LoadLibrary to load the DIGITAL FX!32 agent DLL and then returns. Note that the code built by the enabler must know the location of the LoadLibrary routine in the subject's virtual address space. Fortunately, NT arranges for the system DLLs (including KERNEL32.DLL, which contains LoadLibrary) to be at the same virtual address in all processes on the system. Hence, the enabler can just use the address of LoadLibrary in its own address space. The data written to the subject's memory contains the pointer to LoadLibrary and the full path name of the agent DLL. The enabler then creates a thread of execution in the subject and passes it the address of that data (CreateRemoteThread), raises its priority (SetThreadPriority), and waits for the thread to finish. If all goes well, the subject thread runs and loads the agent DLL into its address space. The agent's main routine is called automatically, and it goes about its work of enabling the subject process.

Inside the subject process, the agent DLL proceeds to hook a number of system functions. It does this by changing the addresses in the image import tables of all loaded modules to point to routines in the agent which replace the system routines. The hooked routines include LoadLibrary, FreeLibrary, CreateProcess, WinExec, LoadModule, and GetProcAddress. Other routines are also hooked to provide an execution environment that makes the system appear to be an x86.

3.2 Running in an Enabled Process

Once a process is enabled, any attempt to execute an image or load a DLL enters the DIGITAL FX!32 agent instead. If the image is an x86 executable, the agent arranges for the DIGITAL FX!32 runtime to take control of execution.

A program executes a new image by calling CreateProcess. Any such call enters the agent's hook for CreateProcess. If the image is a native Alpha executable, the agent passes the request to the system's CreateProcess, enables the new process, and then just lets it run. If the image is an x86 image, the agent adds the name of the runtime to the front of the command line and then creates the process. The DIGITAL FX!32 runtime runs and arranges to load and execute the x86 code. (This has the unfortunate side effect of listing all x86 processes as "fx32" in the processes list of the task manager. We are still looking for ways around this for NT Version 4.0.) Note that since the new process starts in the runtime, it is automatically enabled.

A request to load an x86 library is handled differently. If the current process' main image is an x86 image, the runtime is already present. The hooked LoadLibrary loads the library ("as data" as far as the Alpha NT system is concerned) and starts execution of its main code using either the emulator or translated code. If the current process' main image is a native Alpha image, the runtime is first brought into memory. Note that DIGITAL FX!32 will only load an Alpha DLL into an x86 image if it has enough information to allow it to jacket the calling sequences of all the exported entry points. This is discussed in section 4.1.

3.3 The Root of all Enabling

Any enabled process will ensure that all processes that it creates are enabled. How does this cascade of enabled processes start? By the time a user logs in, all the top-level processes must be enabled somehow, so that any attempt to execute a 32-bit x86 application invokes DIGITAL FX!32.

The processes which must be initially enabled are the Shell (explorer.exe), the Service Control Manager (services.exe) and RPCSS (rpcss.exe). The DIGITAL FX!32 server enables the Service Control Manager and RPCSS when it starts up, usually when the system boots. These two are system processes, and are running even before a user can log in. There is currently a short time window in which the Service Control Manager can attempt to start an x86 service before it is enabled. This causes the x86 service to fail to start. The current work-around is to make the x86 service dependent on the DIGITAL FX!32 server.

Enabling the processes for a logged-in user is trickier. When DIGITAL FX!32 is installed, it stores fx32strt.exe in the registry as the Windows Shell, replacing the real Windows Shell (explorer.exe). When a user logs on, fx32strt runs and creates an enabled version of the Explorer. Thus, by the time the user is logged on, all the top level processes have been enabled. There is one catch to this process. The Explorer checks the registry to see if it is the user's default shell. If so, it runs in a reduced mode, and does not create a task bar or run any programs in the startup group. To get around this, the server temporarily changes the registry's Shell value to point to the Explorer, long enough to fool it into believing (quite rightly) that it is the user's default shell.

4. The DIGITAL FX!32 Runtime

The DIGITAL FX!32 runtime is invoked whenever an enabled process attempts to execute an x86 image. The runtime loads the image into memory, sets up the runtime environment required by the emulator, and then calls the emulator to execute the image.

The runtime duplicates the functionality of the NT loader. This is necessary since the Alpha NT loader will return an error indicating that the image is of the wrong architecture if it is invoked to load an x86 image. Duplicating the functionality of the NT loader requires that the runtime relocate images which are not loaded at their preferred base address, set up shared sections, and process static TLS (Thread Local Storage) sections.

The runtime registers each image it processes with NT by inserting pointers to the image into various lists used internally by the operating system. Maintaining these lists allows the native Windows NT code to correctly implement routines like LoadResource that require access to loaded images. It also means that the DllMain functions of the loaded DLLs are called as appropriate. (The runtime jackets the entry points of x86 DLLs.)

Fortunately, these image lists are in the user's address space and no modification of NT was required to register images with the system. Unfortunately, the structure of these lists is not part of the documented Win32 interface and using them creates a dependency on the version of NT that is being run. This is one of a number of places where FX!32 has dependencies on undocumented features of NT, making it more

dependent on a particular version of the operating system than a typical layered application. On the other hand, it is remarkable that the implementation of FX!32 required no changes to NT.

In addition to being registered with NT, the image is also registered in the DIGITAL FX!32 database. The database maintains the association between the image and the application which uses it. It also returns the name of the translated image to be used with a given x86 image. The database is accessed using an image id obtained by hashing the image's header. The image id uniquely identifies the image by its contents, and is independent of the image name or location in the file system. Both the runtime and the server use the image id to access information about the image which is stored in the DIGITAL FX!32 database.

If there is a translated image in the database, the runtime loads it along with the original x86 image. Translated images are normal NT DLLs, and are loaded by the native LoadLibrary. They contain additional sections holding information required by the runtime to map x86 routines to the corresponding Alpha code.

4.1 Jackets

When the NT loader loads an image, it "snaps" the image's imports using symbolic information in the image to locate the address of the imported routine or data. The DIGITAL FX!32 runtime duplicates this process. However it treats imports which refer to entries in Alpha images specially by redirecting them to refer to the correct jacket entry in the DIGITAL FX!32 DLL jacket.dll.

The "jackets" in jacket.dll are small code fragments which manage the transition between the x86 and Alpha environment. These jackets enable the x86 program to call the native Alpha implementation of the Windows API.

Each jacket contains an illegal x86 instruction that serves as a signal to the interpreter to switch into the Alpha environment. The interpreter calls an Alpha jacket routine at a fixed offset from the illegal x86 instruction. The basic operation of most jacket routines is to move arguments from the x86 stack to the appropriate Alpha registers, as dictated by the Alpha calling standard. Some jacket routines provide special semantics for the native routine being called, as required by FX!32. For example, the jacket for GetSystemDirectory returns the path to the FX!32 directory, rather than the path to the true system directory, so that x86 applications do not overwrite native Alpha DLLs.

More complicated jackets are required in many cases. For instance, many Windows routines are passed the addresses of routines to call back when some event occurs. If these values were to be passed blindly, the Alpha Windows code would make a call to a location containing x86 code, and would certainly crash. A jacket for such a routine is a hand-crafted special jacket which dynamically creates incoming jackets for the procedure-pointer arguments, and passes those to the native Alpha code. When that code calls back to its argument, the incoming jacket enters the runtime to execute x86 code.

The most complicated jacketing problem is associated with OLE. An OLE interface is represented by a table of function pointers. DIGITAL FX!32 jackets these objects' functions in such a way as to allow them to be used from either native Alpha code or from x86 code.

FX!32 provides jackets for entries to over 50 native Alpha DLLs, including jacketing many undocumented routines whose argument lists were determined from the header files in the SDKs.

In order for native Alpha code to interoperate with the x86 environment, it must be possible to jacket the calling sequences for every function call that can cross architecture boundaries. This is possible for system DLLs because their interfaces are (usually) documented. It can be done for DLLs that contain OLE objects because there are strict rules on how those objects publish their interfaces. However, consider the case of an application that is running a native Alpha version, but which accepts plug-in extensions. A plug-in provided as an x86 DLL may have any calling sequence agreed to by the application vendor. DIGITAL FX!32 cannot load such a plug-in unless it is taught how to jacket the interfaces. The current version of DIGITAL FX!32 jackets a few common plug-in interfaces, and we are working on ways to describe arbitrary plug-in interfaces for a future release.

5. The DIGITAL FX!32 Emulator

The emulator has a fundamentally important role in DIGITAL FX!32. It allows x86 applications to run prior to their translation. The first time any x86 image executes under DIGITAL FX!32, it is executed by the emulator.

The emulator also plays an important role as a backup for translated code. In general it is impossible to statically determine all the code that can ever be executed by an application, especially for applications which generate code "on the fly." The emulator provides a mechanism to execute x86 application code which has not been translated. Previous binary translators built by Digital have always depended on the presence of an emulator in this role[1]. A fundamental difference between DIGITAL FX!32 and the earlier binary translators is that large amounts of code are interpreted by the DIGITAL FX!32 emulator the first time an application is run. The performance of the emulator is therefore more of an issue for DIGITAL FX!32 than for the earlier translators.

The DIGITAL FX!32 emulator is an Alpha assembly language program which interprets the subset of x86 instructions that a Win32 application can execute. While an x86 application is running, the state of the x86 processor is kept partially in Alpha registers and partially in a per-thread data structure called the CONTEXT. While in the emulator, a dedicated register always points to the CONTEXT. x86 integer registers are permanently mapped to Alpha registers and the state of the x86 condition codes is maintained in Alpha registers during execution of x86 code. Any part of the x86 state which must be maintained across calls to other parts of the system (for example on calls to Alpha APIs) is stored in the CONTEXT.

The emulator generates profile data for use by the translator while it is interpreting an x86 program. The profile data includes the following information:

- addresses which are the targets of CALL instructions,
- source address, target address pairs for indirect jumps, and
- addresses of instructions which make unaligned references to memory.

The translator uses this information to generate "routines," units of translation which approximate a source code routine. The emulator generates profile data by inserting values in a hash table whenever a relevant instruction is interpreted. For example, when interpreting the CALL instruction, the emulator records the target of the call. When an image is unloaded, either as a result of a call on FreeLibrary or when the application exits, the loader processes the hash table to produce a profile file for the image. The server will process this profile and may invoke the translator to create a new translation of the image.

The emulator uses the same hash table to detect when there translated code is available. When the DIGITAL FX!32 loader brings in a translated image, it builds entries in the hash table that associate the addresses of x86 routines which were translated with the addresses of the corresponding translated code. The loader extracts this information from the translated image. When the emulator interprets a CALL instruction, it looks for the target address in the hash table. If a corresponding translated address exists, the emulator transfers to the translated code.

6. The DIGITAL FX!32 Translator

The translator is invoked by the server to translate x86 images which have been executed by the emulator. As a result of executing the image, a profile for the image will exist in the DIGITAL FX!32 database. The translator uses the profile to produce a translated image. On subsequent executions of the image, the translated code will be used, substantially speeding up the application.

The front end of the translator contains a component called the "regionizer" which divides the x86 image into "routines." Routines are units of translation which approximate real routines in source programs. Each routine is then processed by the other components of the translator to produce Alpha code.

The regionizer uses data in the profile to divide the code in the source image into routines. Each call target in the profile is used to generate an entry to a routine. The regionizer represents routines as a collection of regions. Each region is a contiguous range of addresses which contains instructions that can be reached from the entry address of the routine. Unlike basic blocks, regions can have multiple entry points. The smallest collection of regions which contain all the instructions which can be reached from the routine entry is used to represent the routine. Many routines have a single region. This representation efficiently describes the division of the source image into units of translation.

The regionizer builds routines by following the control flow of the source image. When an indirect jump is encountered while following the control flow, the profile provides a list of possible targets. Without this information from the profile, it would be very difficult to reliably identify the targets of indirect jumps, and they would have to be treated as returns from the

routine. The profile information makes it possible to reliably generate a more complete representation of routines with correct control flow.

The remaining components of the translator process the source image one routine at a time. They build an internal representation of the routine, perform several transformations which produce code more suited to the Alpha architecture, generate Alpha code, and write the resulting translated image.

7. The DIGITAL FX!32 Database

The DIGITAL FX!32 database consists of two parts. The first is a directory tree which contains profile files, translator log files, and translated images. The second part is in the registry and provides information about the x86 applications and images which DIGITAL FX!32 has run on the system.

DIGITAL FX!32 also keeps configuration information in the registry. This information includes things like the maximum amount of disk space to use, the maximum number of images to store in the database, and default translation options (which can be overridden at the application or image level). The registry also holds the work list, which the server uses to schedule translations.

One important piece of configuration information kept in the registry is the DatabaseDirectoryList. This is a list of paths to additional databases which the server can search for image profiles and translation results. Directories on this list are searched the first time an image is executed and can provide information about the image from other machines on the network. This allows DIGITAL FX!32 to use the results of translations performed on other, possibly more powerful machines.

8. The Server

The DIGITAL FX!32 server is an NT Service which normally starts whenever the system is rebooted. The primary role of the server is to automatically run the translator "when appropriate." This makes the overall translation process completely transparent to the user. The server also maintains the database to control DIGITAL FX!32 resource usage.

9. The User Interface

Most of the time, the operation of DIGITAL FX!32 is completely transparent to the user. However, DIGITAL FX!32 consumes system resources and there must be some way for a knowledgeable user to control this resource usage. This is the role of the DIGITAL FX!32 manager. The manager provides a user interface to the configuration information kept in the database.

By interacting with the manager, the user can control various aspects of the operation of FX!32, such as the maximum amount of disk space to use, which information to retain in the database, and when the translator should run.

10. Results

DIGITAL FX!32 had two primary goals: transparent execution of 32-bit x86 applications, and performance that was roughly equal to a high-end x86 platform when running the same applications on a high-performance Alpha system. Both objectives have been meet.

Transparency is provided by the DIGITAL FX!32 agent and a runtime environment which will load and execute an x86 application without a translation step. Applications launch and execute on an Alpha running DIGITAL FX!32 just like they do on an x86.

DIGITAL FX!32 also meet its performance objectives. Figure 1 shows the relative performance on the Byte Benchmark of a 200Mz Pentium Pro and a 500 Mz Alpha running DIGITAL FX!32. For this benchmark, the Alpha running DIGITAL FX!32 provides about the same performance as a 200Mz Pentium Pro. Figure 1 also shows that the Alpha native version of the benchmark runs twice as fast as the Pentium Pro.

Of course, no single benchmark characterizes the performance of a system. However, we have consistently measured performance between a 200Mz Pentium and a 200Mz Pentium Pro for applications running under DIGITAL FX!32 on a 500Mz Alpha.

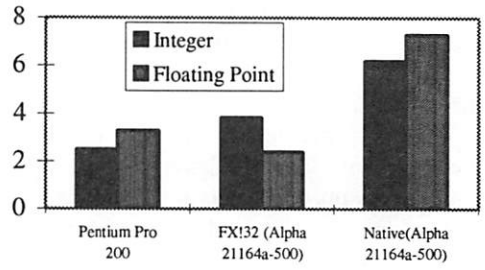


Figure 1
DIGITAL FX!32 Performance on Byte Benchmark

There are some things that the initial version of DIGITAL FX!32 was not designed to do. DIGITAL FX!32 only executes application code. It does not execute drivers, so native drivers are still required for any peripheral installed on an Alpha system. It also fails to provide complete support for x86 services, as discussed in Section 3.3. Another limitation of DIGITAL FX!32 is that it does not support the NT Debug API. Supporting this interface would require that the x86 state could be re-materialized after every x86 instruction, severely limiting optimizations which could be performed by the translator. This limitation is similar to the tradeoff in optimizing compilers where debugging is restricted when optimizations are turned on. Since DIGITAL FX!32 does not support the Debug interface, applications which require it do not run under DIGITAL FX!32. These applications are mostly x86 development environments, and it probably makes sense to run them on an x86 anyway.

Despite these limitations most x86 applications which run on an x86 Windows NT system will run on an Alpha system running FX!32 under Windows NT.

11. Conclusion

DIGITAL FX!32 provides for fast transparent execution of 32-bit x86 applications on Alpha systems running Windows NT. This is accomplished using a unique combination of emulation and binary translation.

12. Acknowledgments

Building a product like DIGITAL FX!32 required a lot of hard work by some extremely talented people. Many of these people contributed the ideas described in this paper. The following engineers were part of the DIGITAL FX!32 development team: Jim Cambell, Anton Chernoff, George Darcy, Tom Evans, Mark Herdeg, Ray Hookway, Maurice Marks, Srinivasan Murari, Brian Nelson, Scott Robinson, Norm Rubin, Joyce Spencer, Tony Tye and John Yates. Charlie Greenman wrote the documentation.

13. Availability

DIGITAL FX!32 is available electronically from

http://www.service.digital.com/fx32

This web site contains more information on DIGITAL FX!32 along with the software itself.

14. References

1. Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson, "Binary Translation", *Digital Technical Journal,* Vol. 4, No. 4, 1992

2. Robert Bedichek, "Some Efficient Architecture Simulation Techniques", USENIX - Winter '90

3. L. Peter Deutsch and Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System", Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, 1983

4. Brian Case, "Rehosting Binary Code For Software Portability", Microprocessor Report, January 1989

5. Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", Technical Report UWCSE 93-06-06, University of Washington, 1993

6. Richard Hank and B. Ramakrishna Rau, "Region-Based Compilation: An Introduction and Motivation", Proceedings of MICRO-28, 1995 IEEE

7. Tom R. Halfhill, "Emulation: RISC's Secret Weapon", BYTE, April 1994

8. Jeffery Richter, Advanced Windows NT, Microsoft Press, 1994

9. Alfred V. Aho, Mahadevan Ganapathi and Steven W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, October 1989

Spike: An Optimizer for Alpha/NT Executables

Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin

spike@vssad.hlo.dec.com

Digital Equipment Corporation

Abstract

Spike is a profile-directed optimizer for Alpha/NT executables that is actively being used to optimize shipping products. Spike consists of the Spike Optimization Environment (SOE) and the Spike Optimizer. Through both a graphical interface and a command-line interface, the Spike Optimization Environment provides a simple means to instrument and optimize large applications consisting of many images. SOE manages the instrumented and optimized images as well as any profile information collected for those images, freeing the user from many tedious and error-prone tasks typically associated with profile-directed optimization. SOE also simplifies the collection of profile information with Transparent Application Substitution (TAS). With TAS, the user invokes the original version of the application and the instrumented or optimized version of the application is transparently executed in its. SOE uses the Spike Optimizer to optimize images. The Spike Optimizer performs code layout to improve instruction cache behavior [Pettis90], hot cold optimization [Cohn96] and register allocation. The optimizations are targeted at large call-intensive applications, where loops span multiple routines, and each routine contains complex control-flow. For this class of applications, Spike provides significant performance improvement, reducing execution time by as much as 20%.

1. Introduction

Profile-directed optimization is rarely used in practice because of the difficulty of collecting, managing, and applying profile information. Spike solves most of these problems, allowing the user to easily optimize large applications composed of many images. In this section we describe the general procedure for using profile-directed optimization, its difficulties, and how Spike overcomes these difficulties.

The first step in profile-directed optimization is to *instrument* each image in an application so that when the application is run, profile information is collected. Instrumentation is most commonly done by using a compiler to insert counters into a program during compilation [Multiflow], or by using a post-link tool to insert counters into an image [Atom, Pixie]. Statistical or sampling-based profiling is an alternative to counter based techniques [DCPI, MORPH]. Some compiler-based and post-link systems require that the program be compiled specially, so that the resulting images are only useful for generating profiles. Many large applications have lengthy and complex build procedures. For these applications, requiring a special rebuild of the application to collect profiles is an obstacle to the use of profile-directed optimization.

Spike directly instruments the final production images so that a special compilation is not required. Spike does require that the images be linked to include relocation information. However, including this extra information does not increase the number of instructions in the image and does not prevent the compiler from performing full optimizations when generating the image.

Most large applications consist of multiple images, a single executable and many dynamically linked libraries (DLL). Instrumenting all the images can be difficult, especially since the user doing the profile-directed optimization may not know all of the images in the application. Spike relieves the user of this task by finding all the DLLs that are used by the application, even if they are loaded dynamically (i.e. with a call to LoadLibrary).

After instrumentation, the next step in profile-directed optimization is to execute the instrumented application and collect profile information. Most profile-directed optimization systems require that the user first explicitly create instrumented copies of each image in an application. Then the user must assemble the instrumented images into a new version of the application, and execute it to collect profile information. As the profile information is generated, the user is responsible for locating all the profile information generated for each image, and merging that information into a single set of profiles. Our experience with users has shown that requiring the user to manage the instrumented copies of the images and the profile information is a

frequent source of problems. For example, the user may fail to instrument each image, or may attempt to instrument an image that has already been instrumented. The user may be unable to locate all the generated profile information, or may incorrectly combine the profile information.

As described in Section 2, Spike frees the user from these tedious and error-prone tasks by managing the instrumented copy of each image as well as the profile information generated for each image.

After profile information is collected, the final step is to use the profile information to optimize each image. As with instrumentation, the typical profile-directed optimization system requires the user to explicitly optimize each image, and to assemble the optimized application. Spike uses the profile information collected for each image to optimize all the images in an application and assembles the optimized application for the user.

2. Spike Optimization Environment

The Spike Optimization Environment (SOE) provides a simple means to instrument and optimize large applications consisting of many images. SOE can be accessed through a graphical interface or a command-line interface that provides identical functionality. The graphical interface, called the Spike Manager, is described in Section 2.1. The command-line interface allows SOE to be used as part of a batch build system such as `make`.

In addition to providing a simple-to-use interface, SOE keeps the instrumented and optimized versions of each image and the profile information associated with each image in a database. When an application is instrumented or optimized, the original versions of the images in the application are not modified; instead SOE puts an instrumented or optimized version of each image into the database. SOE uses Transparent Application Substitution (TAS) to execute the instrumented and optimized version of an application when the user invokes the original version, as described in Section 2.2.

The Spike Optimization Environment allows the user to instrument and optimize an entire application using the following procedure:

1. *Register*: The user selects the application(s) that are to be instrumented and optimized. The user only needs to specify the application's main image. Spike then finds all the implicitly linked images (DLLs loaded when the main image is loaded) and registers that they are part of the application.
2. *Instrument*: The user requests that an application be instrumented. For each image in the application, SOE invokes the Spike Optimizer to instrument that image. SOE places the instrumented version of each image in the database. The original images are not modified.
3. *Collect profile information*: The user runs the original application in the normal way, e.g. from a command-prompt, from the Explorer, or indirectly through another program. Transparent Application Substitution invokes the instrumented version of the application in place of the original version. Any images dynamically loaded by the application are instrumented on the fly. Each time the application terminates, profile information for each image is written to the database and merged with any existing profile information.
4. *Optimize*: The user requests that an application be optimized. For each image in the application, SOE invokes the Spike Optimizer to optimize the image using the collected profile information and places the optimized version of each image in the database.
5. *Run optimized version*: The user runs the original application and TAS substitutes the optimized version, allowing the user to evaluate the effectiveness of the optimization.
6. *Export*: SOE exports the optimized images from the database, placing them in a directory specified by the user. The optimized images can then be packaged with other application components.

2.1. Spike Manager

The Spike Manager is the principal user interface for using the Spike Optimization Environment. The Spike Manager displays the contents of the database, showing the applications registered with Spike, the images contained in each application, and the profile information collected for each image. The Spike Manager enables the user to control many aspects of the instrumentation and optimization process, including specifying which images are to be instrumented and optimized, which version of the application is to be executed when the original application is invoked, etc.

2.2. Transparent Application Substitution

Transparent Application Substitution is a mechanism for transparently executing a modified version of an application, without replacing the original images on disk. SOE uses TAS to load an instrumented or optimized version when the user invokes the original application. With TAS, the user does not need to do anything special to execute the instrumented or optimized version of an application. The user simply invokes the original application in the usual way (e.g. from a command prompt, from the Explorer, or indirectly through another application) and the instrumented or optimized application is run in its place.

TAS performs application substitution in two parts. First, TAS makes the NT loader use a modified version of the main image and DLLs. Second, TAS must make it appear to the application that the original images were invoked.

TAS uses debugging capabilities provided by NT to specify that whenever the main image of an application is invoked, the modified version of that image should be executed instead. In each image, the table of imported DLLs is altered so that instead of loading the DLLs specified in the original image, each image loads their modified counterparts. Thus, when the user invokes an application, NT loads the modified versions of the images contained in the application. Some applications load DLLs with explicit calls to LoadLibrary. TAS intercepts those calls and instead loads the modified versions.

The second part of TAS makes the modified version of the application appear to be the original version of the application. Applications often use the name of the main image to find other files. For example, if an instrumented image requests its full pathname, TAS instead returns the full pathname of the corresponding original image. To do this, TAS replaces certain calls to kernel32.dll in the instrumented and optimized images with calls to hook routines. Each hook routine determines the outcome the call would have had for the original application, and returns that result.

3. Spike Optimizer

The Spike Optimizer is used to instrument and optimize images [Amitabh, Wilson96]. The optimizer is invoked by SOE and can also be invoked directly by the user. There are several phases when instrumenting or optimizing an image. During the first phase, the optimizer finds all of the code contained in an image. NT images mix code and read-only data in the same section, so the optimizer must analyze the flow paths from known code to find as much of the code as possible. If it cannot be determined if part of a section is code or data, it is handled conservatively to preserve correctness.

Next, the optimizer finds all references to addresses that must be updated when parts of the image are moved. These are commonly called relocatable addresses. For Alpha/NT images, these are PC relative branches, data in memory that refer to other data or code, and instructions which load literals that are addresses of code or data. For references to data, the optimizer must also identify the section to which the address refers, so that the address can be changed as the section is moved in memory. For a reference to code, the optimizer must identify the specific instruction that is referenced, because the optimizer can rearrange individual instructions.

After finding all the code, locating all the PC relative branches and the instructions to which they refer is straightforward. Addresses that are data or are literals in instructions can be found because they are pointed to by relocations. Relocations identify code and data that contain addresses that must be adjusted if the system must load an image in to memory at an address other than at its preferred address.

The Spike Optimizer uses a linear list of Alpha machine instructions, annotated with a small amount of additional information, as its intermediate representation (IR). On top of the IR, the optimizer builds a complete compiler-like representation for the image, including a call graph, flow graphs, routines and basic blocks. Images can be very large; for example the largest image in Unigraphics, a CAD application from EDS, contains 36 Mbytes of code in 60,000 routines. Thus, the optimizer's representations must be extremely space efficient.

The Spike Optimizer performs an interprocedural dataflow analysis to summarize register usage within the image [Goodwin97]. This enables optimizations to use and reallocate registers. The interprocedural dataflow is very fast, requiring less than 20 seconds on our largest applications. Memory dataflow is much more difficult because of the limited information available in an executable, so the optimizer only analyzes references to the stack.

Instrumentation or optimization insert, delete, or modify the IR. After instrumentation or optimization is complete, the new IR is output as Alpha instructions and references to relocatable addresses in data are updated to reflect the new layout of the image.

Program	Full Name	Type	Code layout workload	HCO workload
VC (c1)	Microsoft Visual C	compiler front-tend	Compiler test suite	N/A
VC (c2)	Microsoft Visual C	compiler backend	Compiler test suite	5000 lines of C code
SQLSERVR	Microsoft Sqlserver 6.5	database	TPC-C	cached TPC-B
ACAD	Autodesk Autocad	mechanical cad	SDUG benchmark	SDUG benchmark
EXCEL	Microsoft Excel 5.0	spreadsheet	BAPCO	BAPCO
USTATION	Bentley Systems Microstation	mechanical cad	rendering	rendering
WINWORD	Microsoft Word 6.0	word processing	BAPCO	BAPCO
TEXIM	Welcom Software Texim 2.0	Project man-agement	N/A	BAPCO
MPEG	Digital Light & Sound Pack	Mpeg decoder	Mpeg file	N/A
MAXEDA	OrCad MaxEDA 6.0	electronic cad	N/A	BAPCO
PTC	ProEngineer	Mechanical cad	Misc.	N/A
EXCHANGE	Microsoft Exchange	Mail server	Misc.	N/A
VORTEX	SpecInt95	database	SPEC ref	SPEC ref
GO	SpecInt95	game	SPEC ref	SPEC ref
M88KSIM	SpecInt95	simulator	SPEC ref	SPEC ref
LI	SpecInt95	lisp interpreter	SPEC ref	SPEC ref
COMPRESS	SpecInt95	compression	SPEC ref	SPEC ref
IJPEG	SpecInt95	JPEG	SPEC ref	SPEC ref
GCC	SpecInt95	compiler	SPEC ref	N/A
PERL	SpecInt95	interpreter	SPEC ref	N/A

Table 1: Benchmark programs and their workloads

3.1. Instrumentation

The Spike Optimizer instruments an image by inserting counters into the image. Each counter records the number of times a particular piece of code executes. Using these counters, the optimizer can determine the number of times each basic block and control-flow edge in the image executes. Spike uses a spanning-tree technique proposed by Knuth [Knuth73] to reduce the number of counters required to fully instrument an image. For example, in an if-then-else clause, counting the number of times the if and then statements are executed is enough to determine the number of times the else statement is executed as well. Register usage information is used to find free registers for the instrumentation code, reducing the number of saves and restores necessary to free up registers. Instrumentation typically makes the code 30% larger. As part of the profile, Spike also captures the last target of a jump or procedure call that cannot be determined statically. We are adding the ability to collect block counts using statistical sampling with the DCPI continuous profiler [DCPI], which will eliminate the need to instrument an image and will greatly reduce the cost of profiling

Spike's profile information is persistent; small changes to an image do not invalidate the profile information collected for that image. Profile persistence is essential for applications that require a lengthy or cumbersome process to generate a profile, even when using low cost methods like statistical sampling. For example, generating a good profile of a transaction processing system requires extensive staging of the system. With persistence, the user can collect a profile once, and continue to use it for successive builds of a program as small changes are made to it. It is also possible to merge a profile generated by an older image with a profile generated by a newer image.

3.2. Optimizations

Spike performs three optimizations, code layout, hot cold optimization, and register allocation. Profile information is used to guide each optimization.

Code layout reduces the number of misses in the instruction cache and the total number of VM pages touched by the application. Spike uses the Pettis and Hansen algorithm [Pettis90, Hwu89], which has three phases. First, the basic blocks in each routine are rearranged so that frequently executed paths are straight-

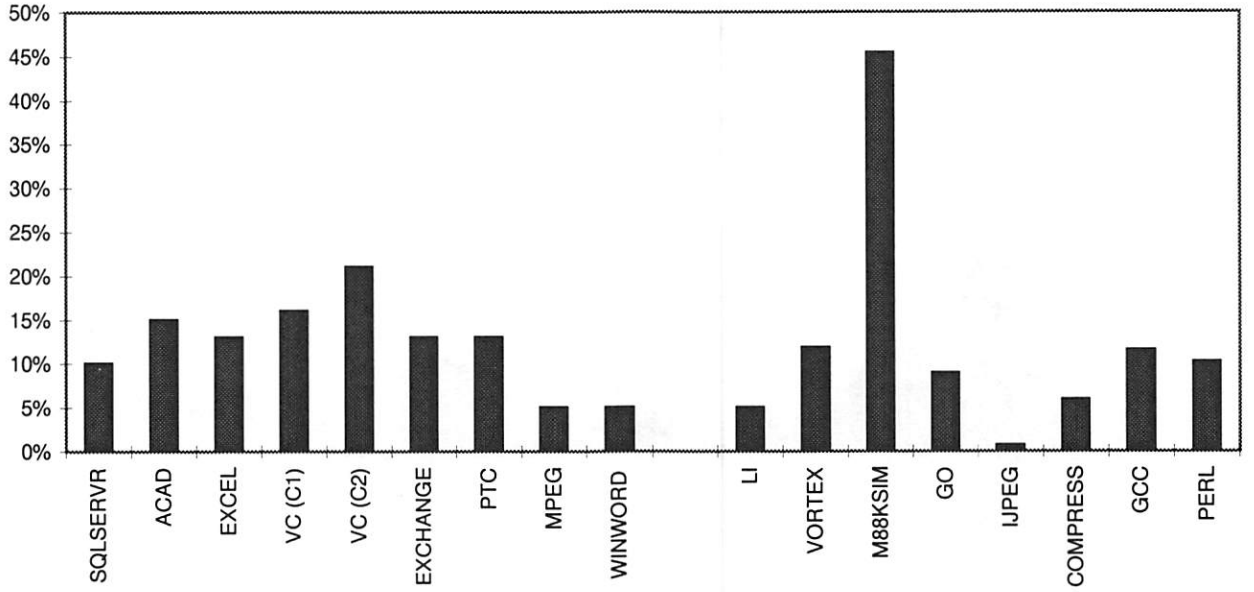


Figure 1: Speedup from code layout

line code; a simple greedy algorithm is used. By reducing the number of taken branches, the processor is able to fetch instructions efficiently and cache lines are better utilized. Next, each routine is separated into a hot and cold section. The hot section consists of the frequently executed basic blocks while the cold section contains the infrequently executed basic blocks. Finally, the hot routines are placed so that frequently called routines are near the caller and cold routines are collected at the end of the image. Splitting and placing routines reduces the chance that routines that call each other will have addresses that clash in the instruction cache.

When examining the hot paths through a routine, it is apparent that many instructions are executed on behalf of the cold paths and are therefore rarely necessary. Hot cold optimization (HCO) [Cohn96] exploits this opportunity. After Spike partitions a routine into hot and cold sections, HCO moves instructions that are dynamically dead or unnecessary in the hot section into the cold section. Stubs are introduced to hold compensation code where necessary.

Many of the opportunities exploited by HCO appear to be poor register allocation decisions, which could be improved with profile information. We are currently implementing a register allocator. Early results show path length reductions of up to 11% in the SPEC95 integer benchmarks.

4. Performance Results

Spike's performance is evaluated using a set of large NT applications that are typical of the applications run on a high performance personal computer, and the SPEC95 [SPEC] integer benchmarks. Table 1 describes each application. All of the programs are compiled with the same highly optimizing backend that is used on Alpha UNIX and VMS systems [Blickstein92].

Figure 1 shows the execution time reduction provided by the code layout optimization. The SPEC95 programs used the training data for profiles and speedups were measured on the reference data. The other programs were trained and measured on the same data. Some experimentation with Excel and VC has shown that the speedup is not very sensitive to training data, as long as it is chosen carefully. Spike speeds up most large applications by at least 5%, and often gets 10% or more. Programs that spend a significant amount of time in inner loops usually get the least benefit, but even the MPEG player has a 5% speedup

Figure 2 shows the speedup for each application after applying HCO, broken down by optimization. The measurements and some of the programs for HCO were done differently from the code layout measurements because they were collected at a different time. As noted in Table 1, the workloads for some of the applications in the HCO measurements were different from the workloads used for code layout. All the programs, including SPEC95, were trained and measured on the same data. The HCO speedups are measured as path

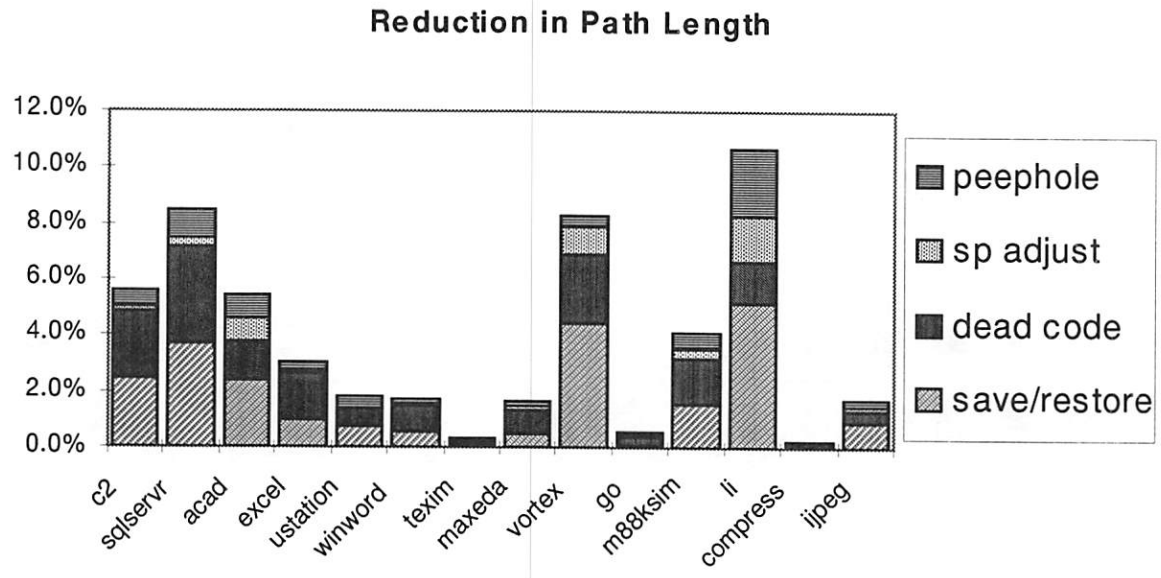


Figure 2: Reduction in path length for HCO broken down by optimization

length reductions, which is the reduction in the number of instructions executed. Using path length reduction allows us measure small effects that would otherwise be obscured by run to run variation. Speedups are in addition to those provided by code layout. For programs that do not spend most of their time in inner loops, HCO usually provides a 5% speedup. About half the speedup comes from removing dynamically dead code, which is moving operations out of frequently executed paths. Most of the rest comes from save/restore, using profile information to improve some register allocation decisions.

5. Summary and Conclusions

Spike is a complete system for optimizing Alpha/NT executables that has eliminated most of the barriers to using profile-directed optimization. Spike provides a simple to use graphical interface, making it easy for a user to instrument and optimize large applications consisting of multiple images. Transparent Application Substitution simplifies profile collection and image management, and persistent profile information enables old profiles to be used even after modifications are made to a program. The optimizations performed by Spike are effective in reducing execution time of large PC applications, producing speedups of 5-20% across a wide range of applications.

6. Acknowledgment

In creating Spike, we were inspired by the elegant solution to the profile-directed optimization problem implemented in FX!32 [Hookway97].

7. References

[Amitabh] A. Srivastava and D. Wall. "Link-Time Optimization of Address Calculation on a 64-bit Architecture," *In Programming Language Design and Implementation*, Orlando, FL, June 1994.

[Atom] A. Srivastava and A. Eustace. "ATOM:A System for Building Customized Program Analysis Tools," *In Programming Language Design and Implementation*, Orlando, FL, June 1994.

[Blickstein92] D. Blickstein, et al, "The GEM optimizing compiler system," Digital *Technical Journal*, 4(4):121-136.

[Cohn96] R. Cohn and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications," MICRO-29, pp. 80-89, Paris, France, December 1996.

[DCPI] http://www.research.digital.com/SRC/dcpi/

[Goodwin97] D. Goodwin, "Interprocedural dataflow analysis in an executable optimizer," To appear in:

Conf. In Programming Language Design and Implementation, Las Vegas, Nevada, June 1997

[Hookway97] R. Hookway, "DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT," *COMPCON*, San Jose, CA, Feb. 1997

[Hwu89] W.W. Hwu and P.P. Chang, "Achieving high instruction cache performance with an optimizing compiler," Proceedings 16th Annual Symposium on Computer Architecture, Jerusalem, Israel, June 1989

[Knuth73] D. Knuth, *The Art of Computer Programming*: Vol. 1, Fundamental Algorithms, Addison Wesley, 1973

[MORPH] http://www.eecs.harvard.edu/morph/

[Multiflow] Lowney et al. "The Multiflow Trace Scheduling Compiler," The Journal of Supercomputing, 7, pp. 51-142, 1993.

[Pettis90] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning" *in Proceedings ACM SIGPLAN Conf. on Programming Language Design and Implementation* '90, pp. 16-27, White Plains, NY, June 1990

[Pixie] MIPS Computer Systems. "UMIPS-V Reference Manual (pixie and pixstats)." Sunnyvale, CA, 1990.

[SPEC] http://www.specbench.org/

[Wilson96] L.S. Wilson, C.A. Neth, M.J. Rickabaugh, "Delivering binary object modification tools for program analysis and optimization," volume 8,1 of *Digital Technical Journal*, pages 18-31

Improving Instruction Locality with Just-In-Time Code Layout

J. Bradley Chen and Bradley D. D. Leupen
Division of Engineering and Applied Sciences
Harvard University
bchen@eecs.harvard.edu

Abstract

This paper describes Just-In-Time code layout (JITCL), a new method for improving the locality of an instruction reference stream by selecting the order of procedures in the text segment during program execution. By determining procedure placement dynamically, this method provides an optimized procedure layout without requiring profile data. For UNIX-style workloads, JITCL provides improvements in instruction cache performance comparable to profile-based layout strategies, while avoiding the requirement of profile data. The late nature of this optimization makes it possible to implement procedure orderings across executable and DLL boundaries, overcoming a limitation of current profile-based techniques. Simulations using Etch [RVL97] on Windows NT show that inter-module JITCL commonly reduces the memory footprint of executable text by 50%.

1. Introduction

Just-In-Time Code Layout (JITCL) is a new method for improving the locality of an instruction reference stream. It achieves similar benefits to profile-based code layout while avoiding the separate profiling step. Current popular methods for procedure layout, such as that described by Pettis and Hansen [PE90], compute an optimized procedure ordering using profile data. Although these schemes can be effective in terms of eliminating instruction cache misses, they share the problems common to profile-based optimization, including profile management and the difficulty in obtaining realistic profiles. JITCL uses a new heuristic which does not require profile information. With this heuristic, an optimized procedure ordering can be computed and applied while the program is running.

An important characteristic of modern Windows applications is their extensive use of dynamically loaded libraries (DLLs). DLLs impose limits on current code-layout optimizations, which cannot typically implement procedure layouts that cross the boundaries of executable files. With JITCL, the optimized procedure ordering is implemented dynamically, with procedures copied into the text segment of the application as it runs. The dynamic nature of JITCL makes it realistic to implement procedure orderings that cross the boundaries created by executable files, introducing the potential for improved instruction cache behavior and smaller working set size for applications that make extensive use of DLLs. This paper describes JITCL and documents its positive impact on instruction cache performance when applied to UNIX and Win32 workloads.

In the next section we describe the JITCL heuristic and compare it to the heuristic described by Pettis and Hansen. We then explain how the JITCL heuristic can be implemented. In Section 3 we present results for JITCL optimization applied within an executable module. Our UNIX results demonstrate that JITCL provides improvements in instruction locality comparable to those commonly achieved with profile-based code layout algorithms for UNIX workloads. Our simulations of JITCL on Windows NT show that it can improve cache performance, avoid bad interactions between modules that occur with the Pettis and Hansen scheme, and substantially reduce text memory requirements.

2. A New Algorithm for Code Layout

2.1 A New Heuristic

In this section we describe the JITCL procedure layout algorithm and how it can be implemented on current hardware. Our algorithm is based on a new heuristic for procedure ordering, which we will present first. The heuristic, which we will call *Activation Order* (*AO*), can be stated simply as:

> *Heuristic (AO): Co-locate procedures that are activated sequentially.*

With AO, procedures are placed in memory in the order they are first invoked. To get some intuition as to why AO is effective and how it compares to current practice,

This work was supported by a grant from the National Science Foundation (CCR-9501365). Additional support for this work was provided by Microsoft Corporation and Intel Corporation.

we compare it to the heuristic used by the popular Pettis and Hansen (*P&H*) procedure layout algorithm:

> *Heuristic (P&H): Co-locate procedures that call each other frequently.*

P&H is applied by generating a weighted call graph, which provides information on the frequency with which procedures invoke each-other, then greedily selecting the heaviest edge from the graph and collapsing the two nodes it connects until the graph is reduced to a single node. The order of procedures in memory is determined by the order in which the nodes are collapsed. Because P&H requires a weighted call graph, it must be computed off-line. In contrast, it is simple to apply AO dynamically, by loading procedures from the executable file into instruction memory in the order in which they are invoked.

A simple example (Figure 1) illustrates the similarities and the differences in the procedure orderings that result from the two heuristics. For this example, assume that the bulk of the computation for this program occurs in the *for* loop. The greedy algorithm used by P&H will start by clustering foo() and bar() around main(), as these are the heaviest edges in the weighted call graph. AO orders the procedures according to the order in which they are invoked. Both heuristics provide the desirable property that the four procedures are clumped together in instruction memory, providing better spatial locality. If the instruction cache is too small to hold all four procedures simultaneously, but is large enough to hold three of them, then the P&H order will generate fewer cache misses than AO because it avoids conflicts between main() and foo(). If the instruction cache is too small to hold three procedures simultaneously, but is large enough to hold foo() and bar(), then AO can give better results. This will occur if the overlap in the instruction cache between foo() and bar() for P&H is larger than the overlap for AO between the code in the for loop of main() and the other two procedures. The effectiveness of each heuristic depends on the frequency with which such phenomena occur in real programs. We explore this issue experimentally in Section 3.1.

2.2 Implementing JITCL

In an on-line implementation of the AO heuristic, procedures are copied from the executable file into instruction memory the first time they are invoked. Our implementation was designed to meet the following goals:

- It should require no special hardware support.
- It should require minimal changes to the operating system.
- It should generate minimal system overhead.

Code	P&H	AO

`void main()`
`{`
`  Initialize();`
`  for (100 iterations){`
`  foo();`
`  bar();`
`  }`
`}`
	foo()
	main()
	bar()
	Initialize()
		main()
		Initialize()
		foo()
		bar()

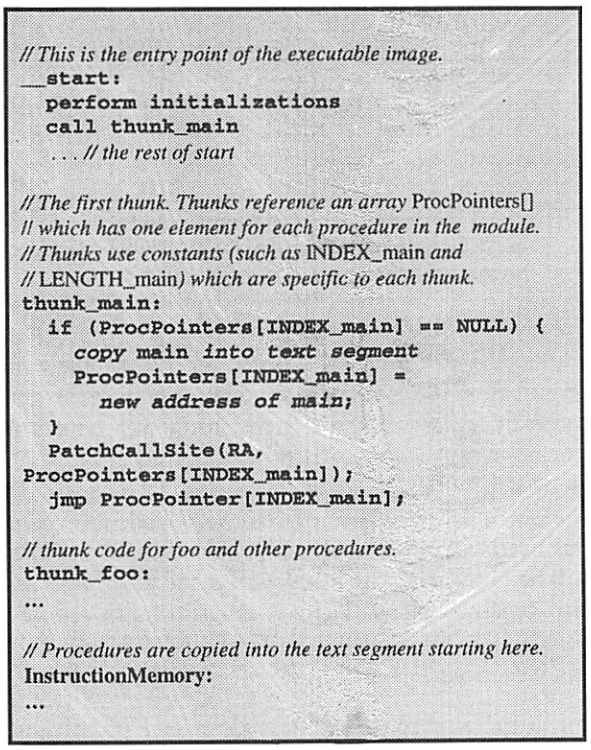
Figure 1. A simple example program.

To implement procedure copying, we replace each procedure call in the original (i.e. conventionally compiled) program with a call to a thunk routine. One thunk is required for each procedure in the program. The task of a thunk is to check if the corresponding procedure has been copied into executable memory (typically from the file-system cache), copying it if it hasn't, and then to patch the call site so that the next time the procedure will be called directly, rather than going through the thunk. A thunk routine finishes by transferring control to the real implementation of the procedure. Figure 2 gives a schematic version of an executable file that has been loaded for JITCL. In the example, the entry point of a program is __start, which after some initialization calls main().

JITCL introduces several new sources of overhead during program execution. One is overhead from maintaining cache-consistency. Note that programs loaded with JITCL are self-modifying. As a result, care must be taken to insure that the instruction cache contents remain consistent with other caches and main memory. Most current machines (including all PCs) implement hardware cache-consistency between the instruction cache and data cache, so no extra operations are required to maintain consistency. When hardware cache consistency is not provided, the routines that patch the call site and copy procedures into the executable segment must provide for instruction cache consistency[1]. In this case, a user-level routine to flush a cache line or set can be implemented with minimal operating system support [CL97]. Just-in-time compilers, as have been proposed for languages such as Java [GJS96], are likely to encounter similar problems.

Each procedure that is referenced dynamically is copied exactly once into instruction memory before being invoked. JITCL relies on file-system caching to minimize the number of operations to disk when copying procedures from the executable file into instruction memory. Assuming reasonable pre-fetching

[1] In the case of PatchCallSite() the cache flush is optional. It may improve performance but does not affect correctness.



```
// This is the entry point of the executable image.
__start:
  perform initializations
  call thunk_main
  . . . // the rest of start

// The first thunk. Thunks reference an array ProcPointers[]
// which has one element for each procedure in the module.
// Thunks use constants (such as INDEX_main and
// LENGTH_main) which are specific to each thunk.
thunk_main:
  if (ProcPointers[INDEX_main] == NULL) {
    copy main into text segment
    ProcPointers[INDEX_main] =
      new address of main;
  }
  PatchCallSite(RA,
ProcPointers[INDEX_main]);
  jmp ProcPointer[INDEX_main];

// thunk code for foo and other procedures.
thunk_foo:
...

// Procedures are copied into the text segment starting here.
InstructionMemory:
...
```

Figure 2: Schematic text segment for a program loaded for JITCL. The executable file includes a text segment as appears above with write+execute attributes. The instructions which implement the actual procedures are found in a separate read-only code segment.

and file-system cache performance, the overhead for procedure copying in JITCL should be comparable to overhead that occurs in systems with demand-loaded text.

Copy overhead is related to the number of distinct bytes of code used by a program. In contrast, the overhead from thunk routines is proportional to the number of distinct locations in code and data that reference procedures and are used during a program run. To gain intuition on these overheads, it is useful to group program runs into several classes:

- *Trivial programs:* For programs that are I/O bound, have small text segments, or that have very short execution times, the impact of any code layout optimization will be small. JITCL will have a small negative impact, and should not be used in this case.

- *Large repeating programs:* For these programs, the benefit of code-layout will often be substantial, and the overhead of copying and thunk-invocation in JITCL will be amortized over a large number of procedure calls. In this case JITCL is beneficial.

- *Large non-repeating programs.* Programs with no locality relative to the instruction cache or memory size will have large instruction reference penalties with or without procedure layout optimizations.

In the next section we demonstrate that the overhead of copying procedures and invoking thunk routines is negligible as compared to the benefit of good procedure layout.

2.3 Refinements

Given the basic framework for JITCL, there are a number of refinements that can be made to improve the basic algorithm. When the text segment is loaded, the invocation order for the first few procedures can be determined statically. For example, given the entry point of __start, it may be possible to determine statically that main() will be the next procedure to be called. In this case main() can be copied statically into the program text segment, instead of relying on a thunk for main() to do the copy at runtime. We expect that the impact of this optimization will be negligible in most cases.

Other refinements involve the use of dynamic information collected during one program run to benefit subsequent runs. For example, the order of invocation of procedures during one run could be used to update the order of procedures in the executable file. In this way, the system could reduce the amount of I/O activity required to copy procedures into memory during later runs.

Once an improved procedure layout has been identified, the executable file could be updated to use the improved ordering statically. In our experience we have found the overhead of procedure copying to be negligible; it is not clear that this refinement will be useful.

A final refinement relates to shared libraries. If library boundaries are ignored when applying JITCL, the result is a system that optimizes procedure layout across executable file boundaries. We explore the impact of cross-module JITCL in our Win32 experiments.

3. Methodology

We used two sets of experiments to evaluate the effectiveness of JITCL. In our UNIX experiments we compared JITCL to Pettis and Hansen code layout. We made this comparison using a memory system model that corresponds to a typical RISC-style system, and UNIX-style benchmarks. We chose this platform to be consistent with the context in which earlier code layout optimizations have been evaluated. The UNIX experiments include a detailed evaluation of the

Benchmark	Description	Text (KBytes)	Time (sec)
Compress	file compression	112	2
Gcc	The GNU C compiler	1552	60
m88ksim	Simulation of Motorola 88K	160	10
Perl	The perl scripting language	376	104
Raytrace	Image rendering	192	18
Xanim	MPEG player	2024	67

Table 1: Unix workloads. All workloads were statically linked.

interaction between instruction and data caches, and the overhead of procedure copying and thunk-routine invocation. Our Win32 experiments build on the UNIX results by exploring the behavior of JITCL for large Windows applications that use many DLLs.

3.1 UNIX Methodology

We evaluated our UNIX workloads on Digital UNIX and the Digital Alpha microprocessor, using the Atom [SE94] instrumentation system to implement our simulator. We instrumented basic blocks, loads and stores in the executable program, inserting calls at these points to our JITCL simulator. Within the simulator, we maintained the state of the caches. We also maintained the additional data structures required by JITCL, and tracked the overhead required to maintain them. Sources of overhead include copy overhead for writing procedures through the data cache, thunk invocation, and data cache traffic from call-site updates.

We implemented a memory system simulation to estimate both instruction and data cache-miss penalties. Our simulated Alpha/UNIX memory system has a split L1 cache with single-cycle latency and no L2 cache. We modeled a 64-bit memory bus operating at 1/3 processor speed, with eleven memory busy cycles to read a 32 byte line. The simulation also used 8K byte coherent direct mapped instruction and data caches, with 32-byte lines. Based on these figures we used a cache read miss penalty of 33 CPU cycles, which corresponds to 11 memory bus cycles. We do not model bus contention or write buffer traffic. JITCL will tend to decrease cache read misses and hence memory traffic. In this respect the results we report are conservative.

Table 1 shows the experimental workloads we used for our UNIX simulations. Many workloads we originally considered did not have interesting instruction cache behavior and were excluded for that reason. We did include one workload, *compress,* with a small instruction cache miss rate. This gives an indication of the impact of the overhead of JITCL in a case where instruction cache behavior cannot be improved.

Benchmark	Instructions (x1000)	L1 I-Cache Miss Rate	L1 D-Cache Miss Rate
Compress	54786	0.0001	0.0204
Gcc	1384160	0.0580	0.3457
M88ksim	542413	0.0441	0.0118
Perl	2157993	0.0442	0.0275
Raytrace	728778	0.0436	0.0119
Xanim	7085956	0.0204	0.0052

Table 2: Summary statistics for UNIX workloads. Instruction counts are in thousands.

Table 2 gives summary statistics for the UNIX workloads. We use several metrics to evaluate the benefit of JITCL: instruction counts, delay cycles, and cache miss rates. Table 2 gives instruction counts for execution of the workloads without optimization. JITCL increases the dynamic instruction count for a given workload, and we use our UNIX workloads to evaluate this overhead in the next section. The benefit of JITCL will be in reducing the instruction cache miss rate, but it also tends to increase the data cache miss rate. For comparison, we provide baseline figures for instruction and data cache misses.

To evaluate JITCL with respect to the best known procedure-layout schemes, we compare results with JITCL layout to results for procedure layout using the Pettis and Hansen algorithm. As JITCL does not use profile information, there is no training input. For the Pettis and Hansen experiments, we trained and tested on the same input. This tends to give a high estimate for the potential benefit of the profile-based optimization; conventional methodology prescribes that different inputs should be used for training and testing [FF92]. The results in Section 4 show that JITCL compares favorably with profile-based layout schemes, in spite of this optimistic estimate of the benefit of profile-based layout.

3.2 Win32 Methodology

We evaluated our Win32 workloads using Windows NT 4.0 on an Intel Pentium-Pro based PC. We used the Etch [RVL97] instrumentation and optimization system to implement an instruction cache simulator. For these experiments, we modeled an on-chip 8K byte 2-way set associative instruction cache, backed by a 512K byte direct-mapped off-chip cache. Both caches used 32-byte lines. We choose to simulate an associative first-level cache as most systems that run Win32 applications use an associative first-level cache. Due to resource issues and our prior investigation of JITCL overheads in the UNIX simulations, we did not include data reference activity in these simulations.

Table 3 shows the experimental workloads used for the Win32 experiments. The Win32 applications are large relative to common UNIX applications. Although the

Benchmark	Description	Text (Kbytes)
Mazelord	Maze game	1445
Window NT perfmon	Display system performance info.	2805
Lotus Wordpro 96	Document preparation	5148
Microsoft Word 7	Document preparation	7694
Microsoft Internet Explorer 3.02	Web browser	4990

Table 3: Win32 workloads. Text size is the total of code size for the executable and all DLLs used directly by the application. As these workloads are interactive we do not give execution times.

Benchmark	Instructions (x1000)	I-Cache Miss Rate	
		L1	L2
Mazelord	5600	0.0239	0.0015
Window NT perfmon	9000	0.0119	0.0006
Lotus Wordpro 96	170000	0.0361	0.0017
Microsoft Word 7	400000	0.0266	0.0007
Microsoft Internet Explorer 3.02	5000	0.0153	0.0014

Table 4: Summary Statistics for Win32 workloads.

size of these applications suggests that code layout optimizations might help them more than the UNIX applications, our analysis will show that other factors, such as cache associativity and interactions between modules, make standard profile-based optimization ineffective for our Windows benchmarks.

Table 4 gives summary statistics for the Win32 workloads. As with the UNIX workloads, the instruction counts in Table 4 do not include overhead instructions introduced by JITCL. The use of the associative rather than direct mapped cache for the Win32 experiments precludes a comparison between the UNIX and Win32 workloads in terms of cache miss rates. Such a comparison is beyond the scope of this paper.

4. Results

4.1 Evaluation of the AO Heuristic

To evaluate the effectiveness of our procedure-layout heuristic, Table 5 compares cache miss behavior for the UNIX workloads and three procedure orderings: the original ordering, an optimized procedure ordering using the Pettis and Hansen algorithm, and procedure ordering using the AO heuristic. Table 5 gives the miss rates for the optimized layouts, as well as the improvement in the miss rate over the original layout.

Table 5 shows that the AO heuristic is effective in improving instruction cache miss rates. AO provides a significant reduction in cache miss rate, on the same order as that for P&H, for all workloads except

	Miss Rate		Improvement	
Benchmark	P&H	AO	P&H	AO
Compress	0.00013	0.00019	(0.00003)	(0.00009)
Gcc	0.05611	0.05474	0.00193	0.00330
m88ksim	0.02172	0.03132	0.02234	0.01274
Perl	0.03701	0.02829	0.00716	0.01588
Raytrace	0.01084	0.01212	0.03272	0.03144
Xanim	0.00292	0.00577	0.01744	0.01459

Table 5: Instruction Cache miss rates (misses/instruction) for the P&H and AO heuristics. Improvement is computed by subtracting the miss rates in the 2nd and 3rd columns from miss rates given in Table 2.

compress. In the case of *compress*, the miss rate is already very low and does not benefit from either AO or P&H. In three of the other five cases, P&H achieves a larger reduction in cache miss rate than AO. These indicate situations where P&H can make effective use of the additional information provided by the profile.

Overall, the results in Table 5 indicate that the AO heuristic can be effective in improving cache miss rates in situations where instruction cache behavior is a significant problem. However, the reduction in cache misses will be beneficial only if they are greater than the overhead of procedure copying and thunk invocation. In the next section we evaluate this overhead.

4.2 Run-time Overhead

Table 6 gives UNIX simulation results for overhead introduced by JITCL. For all the workloads, the number of procedure calls is much higher than the number of call sites (stub calls) or bytes of code copied to support JITCL. Even for *compress*, the overhead of JITCL is amortized over a sufficiently large period of activity that it less than 0.05%.

JITCL also generates additional data cache traffic, as procedures are copied to instruction memory through the data cache, and instruction cache traffic, due to activity from thunk and copy routines. Table 7 quantifies these effects, giving the change in the instruction and data cache miss counts for our experiments. Table 7 also gives the increase (or decrease) in cache miss rate that occurs due to JITCL. In all cases the increase in data cache miss rate is very small (less than 0.001). Table 7 also shows the positive impact that JITCL can have on instruction cache performance.

	Calls (x1000)	Stub Calls	Bytes Copied	Instruction Overhead	Overhead (%)
Compress	923	85	44672	17000	0. 031
Gcc	19647	6644	1091856	1328800	0. 096
m88ksim	7588	240	57744	48000	0. 009
Perl	29472	528	248064	105600	0. 005
Raytrace	89490	354	91920	70800	0. 010
Xanim	15570	1562	353536	313400	0. 060

Table 6: Overhead of dynamic code movement - Code Movement. Overhead is computed as (100 * overhead instructions / original instruction count).

Benchmark	Additional I-Cache misses	Additional D-Cache Misses	Increased I-Cache Miss rate	Increased D-Cache Miss rate
Compress	4717	2877	0.0001	0.0001
Gcc	(4484481)	83765	(0.0032)	0.0001
m88ksim	(6907881)	4650	(0.0127)	0.0000
Perl	(34256776)	16899	(0.0159)	0.0000
Raytrace	(22911716)	6319	(0.0314)	0.0000
Xanim	(103351269)	24583	(0.0146)	0.0006

Table 7: Overhead of dynamic code movement - Cache penalties. This table shows the increase (decrease) in cache misses and cache miss rates that occurs with JITCL.

4.3 Overall Impact of JITCL for UNIX workloads

Figure 3 gives results for the overall impact of JITCL for the UNIX workloads. We estimate the benefit of JITCL in terms of cycles saved per instruction, using the cache miss penalties given in Section 3. Figure 3 shows that JITCL has a comparable benefit to profile-based procedure layout schemes such as P&H. JITCL is not beneficial for workloads such as *compress* where code layout is not a problem, but even in these cases, the overhead of JITCL is small enough to be insignificant.

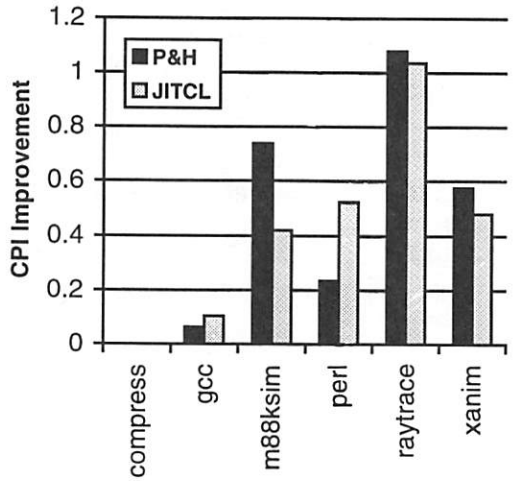


Figure 3: CPI Improvement for UNIX Workloads.

4.4 Cache Effects of JITCL for Win32 Workloads

Figures 4 and 5 show instruction cache miss rates for the Win32 Workloads. Due to cache associativity and the large memory requirements of these programs, JITCL does not provide a consistent significant improvement over the default procedure ordering in the first-level cache. The behavior for P&H layout is consistently worse than for the other procedure orderings. The problem with P&H is that it neglects interaction between modules. P&H improves locality within a module by spreading the active procedures in a module evenly over the cache. However, the improved layout within modules also increases competition between modules. The overall result is that P&H procedure layout is not beneficial for the Win32 benchmarks.

For the larger second-level cache, JITCL provides somewhat better cache miss rates than the default cache layout, whereas performance for P&H is consistently worse. Although the effectiveness of JITCL for improving the cache miss behavior of these workloads is limited, the next section shows that it can provide a substantial benefit in reducing program working set size.

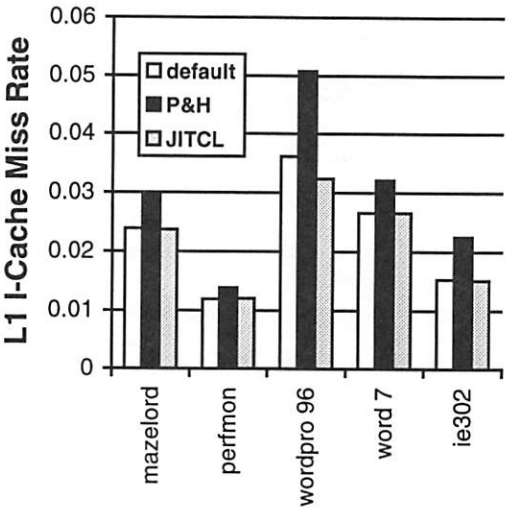


Figure 4: First Level Cache Miss Rates for Win32 Workloads.

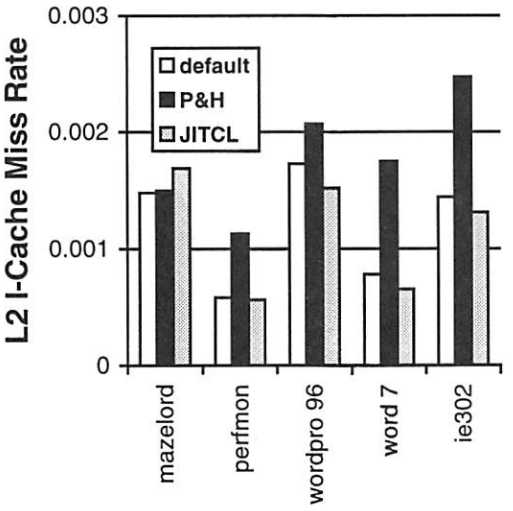


Figure 5. Second Level Cache Miss Rates for Win32 Workloads.

4.5 Working Set Size of Win32 Benchmarks

Figure 6 shows memory space occupied by executable program text for the Win32 programs, with and without JITCL procedure layout. JITCL only requires memory for the procedures that are actually used during a run of the application. As a result, JITCL commonly reduces executable memory requirement by about 50%. This can be a substantial benefit for decreasing the memory footprint of an application without sacrificing functionality.

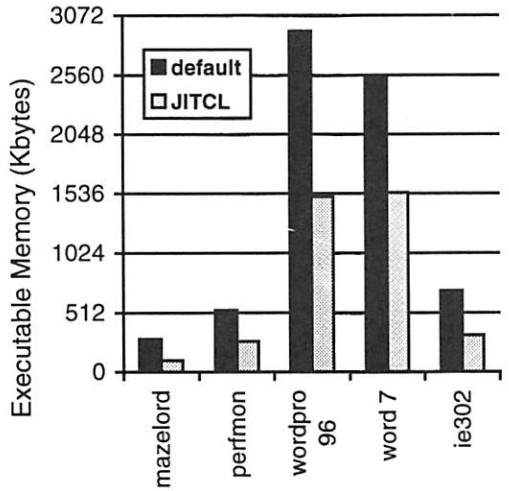


Figure 6: Executable Memory Requirements for Win32 Workloads, in 4k byte pages.

Pettis and Hansen layout also has benefits in terms of reducing program memory requirements. However, a comparison of JITCL and P&H for reducing memory requirements is difficult. Given a fixed training/testing input, JITCL and P&H layouts will require a comparable number of memory pages, as both heuristics cluster the procedures used by the program in memory. For multiple inputs, however, the Pettis and Hansen layout will typically give worse performance, as the procedures used during training runs will not exactly match the procedures invoked during an actual use of the program. As a result, JITCL will tend to be more effective a reducing program working set size and improving overall performance.

5. Conclusions

We have described JITCL, a code layout optimization improving instruction cache behavior through dynamic code layout. The optimization is based on a simple heuristic that places procedures in the order in which they are called. This heuristic permits an on-line implementation, avoiding the need for training and profile information. For UNIX workloads, JITCL achieves a comparable benefit to popular profile-based procedure layout schemes without requiring profile information. Although the instruction cache benefits for our Win32 experiments are not significant, the reductions in memory requirements are typically about 50%, making JITCL interesting as a technique for reducing program working-set size. Although JITCL introduces some overhead, our experience indicates that the overhead is negligible compared to the benefit of improved procedure ordering.

Acknowledgements

This work was supported by a grant from the National Science Foundation (CCR-9501365). Additional support for this work was provided by Microsoft Corporation and Intel Corporation.

References

[CL97] J. Bradley Chen and Bradley D.D. Leupen. "Improving Instruction Locality with Just-In-Time Code Layout," Technical Report, Division of Engineering and Applied Sciences, Harvard University, March 1997.

[RVL97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Brian Bershad, Hank Levy, and Brad Chen. "Etch, an Instrumentation and Optimization tool for Win32 Programs." *Proceedings of the 1997 USENIX Windows NT Workshop*, USENIX Association, Berkeley CA. (*In this volume*). See also http://www.cs.washington.edu/homes/bershad/Etch/.

[FF92] J. A. Fisher and S. M. Freudenberger. "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 85-97, October 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison Wesley, Reading, MA, 1997.

[PH90] K. Pettis and R. Hansen. "Profile Guided Code Positioning," *Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation*, ACM, pp. 16-27, June 1990.

[SE94] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 196-205*, June 1994. See also DEC WRL Research Report 94/2.

The RTX Real-Time Subsystem for Windows NT

Bill Carpenter, Mark Roman, Nick Vasilatos and Myron Zimmerman
VenturCom, Inc.
215 First St.
Cambridge, MA 02142
{carp, marcus, boxer, myron}@vci.com

Abstract

This paper describes a subsystem for the Windows NT 4.0 Operating System which implements a kernel-mode execution environment for Win32 compatible tasks and threads that have hard real-time performance characteristics (deterministic interrupt response and dispatch latencies). This subsystem is a proper OS extension which requires no modifications to the standard OS kernel and limited modifications to the NT Hardware Abstraction Layer (***HAL***). This gives the motivation for the approach, describes the design and evaluates the success of the implementation in the context of other strategies for extending general purpose OS kernels.

1. Introduction

Real-time features have been moving from special purpose operating systems into general purpose operating systems for at least 10 years. VenturCom has produced a succession of real-time UNIX™ versions going back as far as 1984. Other implementations are described in [Furht et al. 91 and IEEE 93].

These were responses to the extremely reasonable desire to leverage the features of the underlying general purpose operating system in real-time applications development. This included both accessing the rich feature set of the general purpose OS and accessing available off the shelf applications, utilities and services in developing the particular real-time application at hand.

This is even more so with Windows NT based applications. The system services are rich, diverse and wildly popular with the programming cognoscenti. The available base of applications is vast and growing rapidly.

This is about how real-time computing can be facilitated for a particular general purpose operating system – Windows NT 4.0 with the addition of a subsystem that supports the execution of very low latency real-time threads with Win32 compatible system services. This approach reflects lessons learned about the costs of massive source code whacking – and the benefits of modularity vis a vis the separation of functionality in systems engineering.

2. Real-time Extensions

Requirements for real-time extensions were developed in a process of consultation with company partners whose applications are concentrated in industrial automation and telephony domains. These however span nearly the full range of functionality traditionally associated with hard real-time systems. Windows NT 4.0 has significant real-time features but gating limitations as well [Sommer 96, VENTU 96, Timmerman & Monfret 96, MICRO 95]. The problems most often cited are priority inversion, the paucity of real-time thread priorities and unbounded system response times.

It was further clear from the consultation process that real-time extensions to the system should conform as closely as possible to the standard interfaces of the host OS. It was therefore stipulated that operations in common between real-time objects and normal objects should be accessible via a common interface.

This lead to the definition of a subset of the Win32 API which includes all basic execution control, memory management, communications, synchronization, I/O and configuration operations to be supported by the added real-time threads and to the definition of added interfaces for real-time operations (stack/heap pre-allocation and locking, interrupt and i/o port attachment, clocks and timers) to be provided for both real-time and normal objects. The result is a unified – Real-time Application Programming Interface (RTAPI) with which application elements intended for both the normal Win32 environment and the extension real-time environment can be designed – and with which design and code can be shared.

The implementation of these extensions spans two components. The first implements the real-time operation

extensions for normal Win32 objects. The second implements the Real-Time Subsystem (***RTSS***). Both exploit limited modifications developed for the NT HAL (which primarily provide extensions for clock and timer device programming and interrupt management).

Figure 1 illustrates how the Real-Time Subsystem interfaces to NT. The HAL and the RTSS driver provide the Real-Time Application Interface (RTAPI) to real-time processes, which are linked and loaded as NT drivers. The difference between an NT driver and the real-time process is that the real-time process uses RTAPI calls instead of the NT device driver interface.

2.1 Thread Scheduling

The real-time thread manager offers 128 thread priorities and controls priority inversion. The threads are scheduled by priority and within a priority, in round robin order. There is no sharing of processors based on a fixed time slice. All RTSS threads must give up the processor by waiting, changing thread priority or otherwise completing execution. All RTSS threads run before any NT threads can run.

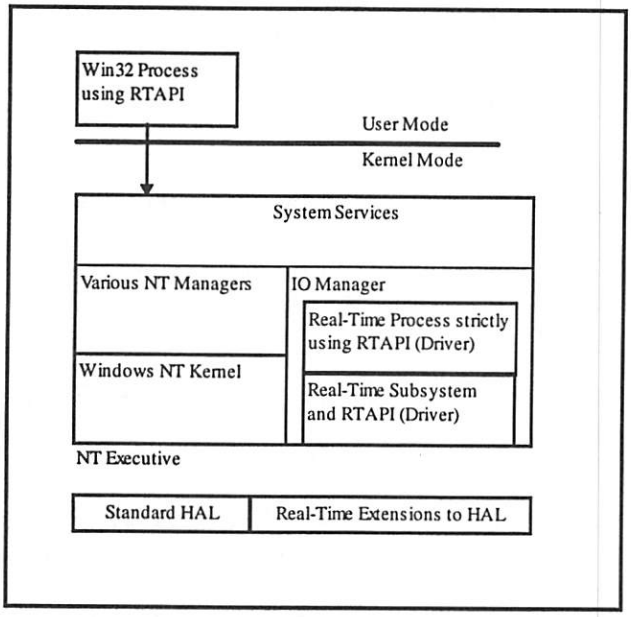


Figure 1 Real-Time Subsystem

The thread manager gets control of the processor in response to interrupt processing by the HAL. Two of these interrupts, the clock and the RTSS software interrupt are fixed interrupt sources. The clock interrupt handler handles timer expirations and the RTSS software interrupt causes the RTSS to examine the queue of messages from NT. The last type of interrupt is from a device for which an RTSS process registered an interrupt handler.

The net effect of these interrupts is that some number of RTSS threads will become ready to run. When all of the threads have finished their immediate work, the thread manager becomes idle, the RTSS switches back to the HAL stack and normal NT processing occurs.

2.2 Interrupt Handling

Interrupt objects are used to claim system interrupt resources and handle interrupt events. The interrupt object is an abstract type which captures the resource usage and the service routine associated with an interrupt.

All interrupt service routines in RTSS are realized as real time threads. In designing the system in this way, a small performance tradeoff was made. For a given interrupt, the handler latency increases to include thread switching time. This is a relatively small increase in latency, however. The benefit to this approach is that all real time system activities are captured as real time threads, which provides a simple, coherent approach to setting priority across the real time subsystem, and allows for a more robust and verifiable implementation of the subsystem.

Interrupt masking is achieved through lazy or optimistic interrupt masking. This technique manages the interrupt level in software to reduce the overhead of hardware programming. Actual hardware masking occurs only in the event of an actual hardware interrupt, at which point the mask is set to prevent further interrupts, and the hardware event is noted so that it can be dealt with at the end of the critical code section. This technique is described in [Stodolsky et al. 93].

2.3 Communication with NT

In order for NT and the RTSS to communicate, two message queues are maintained by the RTSS. One queue is for messages passing from NT to the RTSS and the other queue is for messages passing from the RTSS to NT. The queues are implemented as circular buffers of messages. This organization is necessary because at this lowest level, NT and the RTSS are not synchronized except for the ability to atomically write the index of the last message written to the queue.

RTSS processes request NT services through this channel. Although the NT device driver interface is in the same address space of the RTSS process, the process must not call these interfaces. This restriction is neces-

sary because the RTSS thread must wait at the message queue instead of being suspended by the NT executive.

3. RTSS Objects

Beside thread performance, the most important feature of the RTSS is the duplication of NT objects and the NT object technology. The NT object technology is described in [Custer 1993]. Adhering to the NT object technology gives Win32 fluent programmers immediate familiarity with the RTSS objects and interface.

For instance, the RTSS implements a real-time semaphore. The RTSS semaphore calls are listed in the table below with along with their Win32 equivalents.

Table 1 Semaphore Call Comparison

RTAPI Semaphore Calls	Win32 Semaphore Calls
RtCreateSemaphore	*CreateSemaphore*
RtOpenSemaphore	*OpenSemaphore*
RtReleaseSemaphore	*ReleaseSemaphore*

The RtCreateSemaphore call creates a semaphore and returns an RTSS object handle to the RTSS process. RTSS processes have a process specific object table. The RTSS object handle is an index into the object table which contains the pointers to an instance of an RTSS object.

The RTAPI calls also contain the generic object functions such as *RtCloseHandle*, *RtDupHandle*, and for objects having a signaled state, *RtWaitForSingleObject.*

The RTSS maintains other features of the NT object technology, such as name space and object retention. However, in order to keep our development effort within reasonable limits, we have not implemented resource accounting or protection. The *RtCreateXxx* calls retain the *LPSECURITY* arguments found in their Win32 counterparts so that security may be added in the future without changing the interfaces.

While leaving out the security and resource accounting simplifies the RTSS object technology, the RTSS has an additional feature not found in NT systems, which is that a class of the RTSS objects are remotely accessible from the Win32 environment and that some NT objects are remotely accessible from the RTSS environment.

Remote access to an object depends upon which of the following categories an object belongs.

3.1. Subsystem Specific Objects

In terms of remote access, a subsystem specific object is not remotely accessible by processes in the other subsystem. The prime example of this type of object is the thread and objects derived from the subsystem threads, the RTSS timer and interrupt objects. This not a serious restriction since other process' threads are not accessed except when a debugger controls a process. In the case of the two subsystems under consideration, debugging techniques are drastically different.

Denying remote access to the subsystem specific objects does not limit the usefulness of interfaces in either environment. This is the case with the interrupt handler object. The interfaces, *RtAttachInterruptVector* and *RtReleaseInterruptVector* were implemented first in the Win32 environment and later in the RTSS environment. The RTSS interrupt object gives deterministic response while the Win32 interrupt object allows easier debugging.

3.2. NT System Objects

The NT system objects are those objects to which the RTSS processes have remote access. The typical object here is the file system object. Interfaces to these objects are supported by a local RPC which passes over the communication channel described in section 2.3. The *RtCreateDirectory* call is one such example.

Because the NT endpoint is in the RTSS driver, the driver must perform the further step of translating the remote *RtCreateDirectory* into the NT device driver interfaces. As noted earlier, these calls are currently in the address space of the RTSS process, however, calling these directly will lead to scheduling disaster in the real-time subsystem.

This category of remote access easily admits equivalent functionality in the Win32 environment. For instance, the Win32 version of *RtCreateDirectory* is simply wrapper around the Win32 *CreateDirectory* call.

3.2. Shared Objects

This is the most interesting class of objects. The shared objects provide synchronization and communication between Win32 and RTSS processes. There are three objects available: the semaphore, the mailslot and the shared memory object. The semaphore and mailslot are implementations of the Win32 objects. The intent of adding a shared memory object and interface is to lessen the complexity associated with creating shared memory using the Win32 calls *CreateFile* and *MapViewOfFile*.

Win32 and RTSS applications have identical access to these objects through the RTSS driver and across the NT-RTSS communication channel. In fact, two Win32 programs can perform synchronization with an RTSS semaphore in exactly the same way as two processes would use a Win32 semaphore. This also means that a Win32 program that restricts itself to using only the RTAPI interface will synchronize without modification when it is compiled with the RTSS libraries and executed in that environment.

Using the mutex object admits unbounded priority inversion into the system when a Win32 process must run in order to release the mutex. We expect that the typical use of this class of object will be either the shared memory or the mailslot. We also expect that priority inversion by NT and starvation of NT will not be an issue in dual and multi-processor systems where at least one processor always runs NT.

4. Conclusions

To be honest, large scale modifications to the NT OS kernel were not an option since the source code is not distributed by the vender. That notwithstanding, our total effort to implement the features described is not dramatically larger for having produced a well isolated subsystem than it would have been for having made corresponding modifications to the facilities of the host OS. Furthermore, our maintenance burden going forward will be reduced dramatically.

Our implementation benefits as well from NT's consistent object semantics which provide useful guidance on how new objects or alternate implementations of existing objects should proceed. The effort to stay as compatible with Win32 as possible makes the features of the RTSS attractive to Win32 programmers and reduces the burden of dividing an application into cooperating components for host and RTSS execution environments to very acceptable levels.

4.1. Future Directions

First customer applications based on the current system have been developed and are being readied for deployment. These require manual verification of system/application timing characteristics. There is rich literature on real-time scheduling which we plan to exploit for automating the scheduling design for application components on the RTSS. Approaches where a priori processor requirements are known (deadline scheduling – [Sommer & Potter 96 and Tokuda et al. 90]) or adaptive schedule generation based on dy-namic system state such as in [Jones et al 96] are un-der consideration.

There are no provisions for process partitioning and protection in the current implementation. There is literature on safely allowing untrusted programs to execute in protected environments such as the kernel address space. How the VINO kernel protects itself is described in [Seltzer et al. 96] and an overview is given in [Small and Seltzer 96]. Another way of creating a safe environment would limit the RTSS run-time environment to a safe language such as JAVA.

References

[Custer 93] H. Custer, "Inside Windows NT." Redmond, Washington, Microsoft Press, 1993.

[Furht et al. 91] B. Furht, D. Gorstick, D. Gluch, G. Rabbat, J. Parker and M. McRoberts, "Real-Time Unix® Systems, Design and Application Guide" Kluwer Academic Publishers, Boston 1991.

[IEEE 93] IEEE, "Portable Operating System Interface (POSIX™) Part 1: System Application Interface Amendment 1: Realtime Extension." IEEE, 1993.

[Jones et al. 96] M. Jones, J. Barrera, A. Forin, P. Leach, D. Rosu and M. Rosu, "An Overview of the Rialto Real-Time Architecture." In Proceedings of the Seventh ACM SIGOPS European Workshop, September, 1996.

[Khanna et al. 92] S. Khanna, M. Sebr\gr{e}e and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0." In Proceedings of the USENIX Winter 1992 Technical Conference, January, 1992.

[MICRO 95] Microsoft, "Windows NT and Real-Time Operating Systems" Available at `http://www.microsoft.com/kb/articles/q94/2/65.htm`, 17 Jan. 1995.

[Seltzer et al. 96] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions." In Proceedings of the USENIX Second Symposium on Operating Systems Design and Implementation, Seattle, October, 1996.

[Small & Seltzer 96] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies." In Prodceedings of the USENIX 1996 Annual Technical Conference, January, 1996.

[Stodolsky et al. 93] D. Stodolsky, J. Chen and B. Bershad, "Fast Interrupt Priority Management in Operating System Kernels." In Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures, September, 1993.

[Tokuda, et al. 90] H. Tokuda, T. Nakajima, P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," In Proceedings of the Usenix First Mach Symposium, October, 1990.

[Sommer 96] S. Sommer "Removing Priority Inversion from an Operating System. Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96), January, 1996.

[Sommer & Potter 96] S. Sommer and J. Potter, "An Overview of the Real-Time Dreams Extensions." In Proceedings of The Third Australasian Conference on Parallel and Real-Time Systems (PART'96), September, 1996.

[Timmerman & Monfret 96] M. Timmerman and J. Monfret, "Windows NT as Real-Time OS?" From Real-Time Magazine and reprinted at `http://www.realtime-info.be/encyc/magazine/articles/winnt/winnt.html`, 1996.

[VENTU 96] VenturCom, "High Frequency Clock and Timer Facilities." Available at `http://www.vci.com/prod_serv/nt/interim.html`, 1996.

A Scheduling Scheme for Network Saturated NT Multiprocessors

Jørgen Sværke Hansen Eric Jul

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen, Denmark
E-mail: {cyller,eric}@diku.dk

Abstract

The use of high performance networking technologies, e.g., ATM networks, demands much from both operating systems and processors. During high network loads, user threads may be starved because the processor spends all its time handling interrupts.

To alleviate this problem, we propose using a two level network interface servicing scheme that uses interrupts during low network loads to provide low latency, and polling threads during high network loads to avoid user thread starvation.

In this paper, we examine the use of such a scheme on dual-processor workstations running Windows NT connected by an ATM network. Performance evaluation of our multiprocessor prototype implementation shows that using our two level scheme can improve performance when used carefully.

1 Introduction

High performance networks based on, e.g., ATM often demand substantial processor time; so much that processors can become saturated with network traffic leaving little or no time for actually processing data.

As part of a project concerning ATM network striping[1], we have considered how to efficiently handle multiple network interfaces. Processing the data that arrives on just a single high-speed network interface is a problem even for fairly high performance workstations; using several high-speed network interfaces (as we will be doing when performing network striping) will only aggravate the situation. Such processor overload can be handled by using extremely powerful processors. However, there are both economical and physical limits on how fast a processor that it is possible to use. As an alternative we propose using multiprocessors to provide sufficient processing power.

Previous work in the area of multiprocessor network performance has concentrated mainly on improving the performance of higher level protocols ([1], [4] and [5]), and furthermore, these approaches use a single network interface. We consider the scheduling issues related to handling one or multiple network interfaces on multiprocessors.

In the following, we first describe the problems of thread starvation, then we present a two level network interface servicing scheme that uses interrupt-driven servicing at low network loads, and polling threads at high network loads. Lastly we evaluate the performance of the two level scheme.

We have implemented this scheme on dual-processor workstations running Windows NT connected by an ATM network. As network interfaces, we use Olicom ATM network interfaces, and Olicom A/S has provided us with access to the source code for the ATM network interface drivers.

2 User Thread Starvation

In interrupt-driven kernels the total processor usage of a network application can be split into two parts, a part used by the user threads, and a part used by the interrupt-routine. Ideally, the partitioning should be such that the interrupt-routine delivers packets at the rate in which the user thread consumes them. To avoid packet loss in case of timing mismatch between user thread and interrupt-routine, a limited number of packets may be buffered in the I/O subsystem untill the user thread collects them. As long as the total demand for processing power does

[1] For more information visit the project homepage at http://www.diku.dk/distlab/Research/CIT/SPAN/span.html.

not exceed what is available, this should cause no problem.

When processing power is a problem during heavy network loads, the threads on the system are starved due to the fact that interrupt-routines have absolute priority over any other thread in the system, and thus the bulk of processing time is used processing interrupts from the network interfaces receiving data. The actual consumers of the received data are not allowed to process the data, and this may cause upper layer buffers in the network subsystem to overflow. The result can be that a large amount of processing power is used on receiving data that is subsequently discarded. Mogul and Ramakrishnan [3] identified this problem and propose to avoid this situation (which they call *receive livelock*) by using interrupt-initiated polling. When a network interface issues an interrupt, its handler merely starts a polling thread. This thread is scheduled together with any other threads in the system, thus reducing the livelock problem. Another benefit from using a polling thread is that the number of interrupts and context switches per received network data unit is lowered in stress situations.

The thread starvation problem is not necessarily removed by adding more processors to an interrupt-driven system. Because an interrupt steals processing time from the thread that it is interrupting, there is the danger of starving a thread during heavy network load. On a multiprocessor, a situation (which we call *thread pinning*) may arise where some processors are almost idle, while another processor is heavily loaded servicing the network interfaces and starving the thread that was to process the incoming data. Furthermore, in the case where a multithreaded user application is used, the user thread workload per packet may be larger than that of the interrupt-routine. This may produce suboptimal performance as illustrated in figure 1. The figure shows how the processor usage is divided between user threads, interrupt-routine and idle time on a two processor machine. As full load is reached, the interrupt-routine starts stealing processor time from the user threads. This continues until the processor usage of the interrupt-routine reaches the capacity of a single processor. If the multiple threads are sharing data, the interrupt-routine might further degrade performance, if it interrupts a thread holding a lock to shared data. As illustrated in figure 2, this will occur when the total load generated by the user threads exceeds the capacity of one processor. From that point the user threads, that are sharing processor with the interrupt-routine, may be interrupted, and thus result

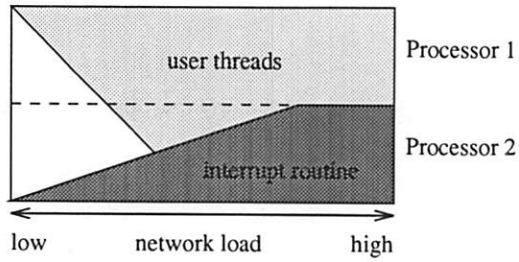


Figure 1: Processor usage as a function of network load on a two processor machine in the case with suboptimal performance during high load due to interrupt-handling.

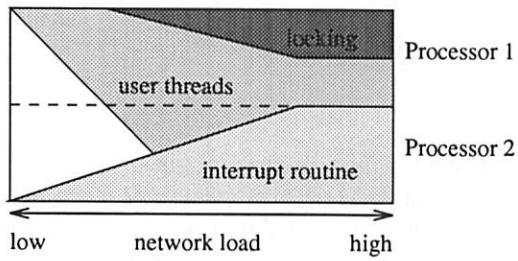


Figure 2: Processor usage as a function of network load on a two processor machine illustrating the performance degradation caused by user threads waiting on locks held by an interrupted thread.

in that threads running on the other processor must wait for a lock held by the interrupted thread. In the figure the idle time resulting from this is marked with *locking*.

To alleviate these problems, the scheduling of network handling on multiprocessors needs to be considered. One possible solution to the thread pinning problem is that the interrupt-handling routine at regular intervals yields the processor, thus allowing the user thread to be rescheduled—possibly on a less loaded processor. Another solution is to utilize that some multiprocessor architectures (e.g., the Pentium Pro) have support for controlling which processor is to receive a given interrupt on the basis of a priority assigned to each processor. As long as there is only one active thread handling network data, and as long as there are fewer network interfaces than processors, this would alleviate the thread pinning problem. Alternatively, one might consider simply disabling interrupts from the network interface(s) causing the overload, but this might hinder the progress of the user threads in the case where they depend on sending data as part of their processing. Lastly, a scheme with a polling thread as described in [3] can be used. By letting one or more threads handle the network interfaces, the usual scheduling mechanism of the operating system may be used to distribute the load on the available processors. This should be

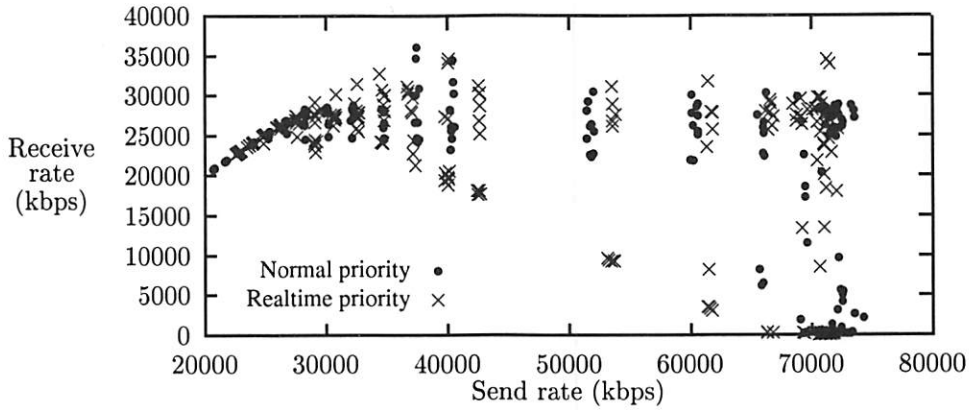


Figure 5: Variation in network throughput with the purely interrupt-driven system. Each dot in the graph represents a single measurement with 1024 bytes UDP packets.

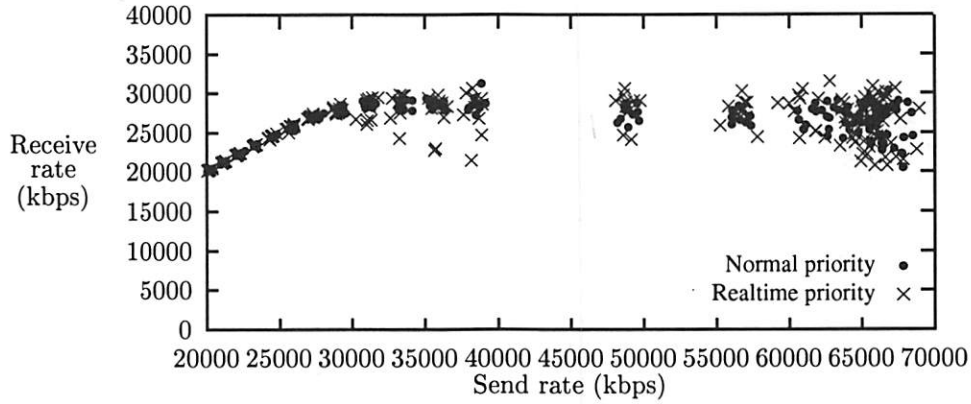


Figure 6: Network throughput with the two level system. Each dot in the graph represents a single measurement using 1024 bytes UDP packets.

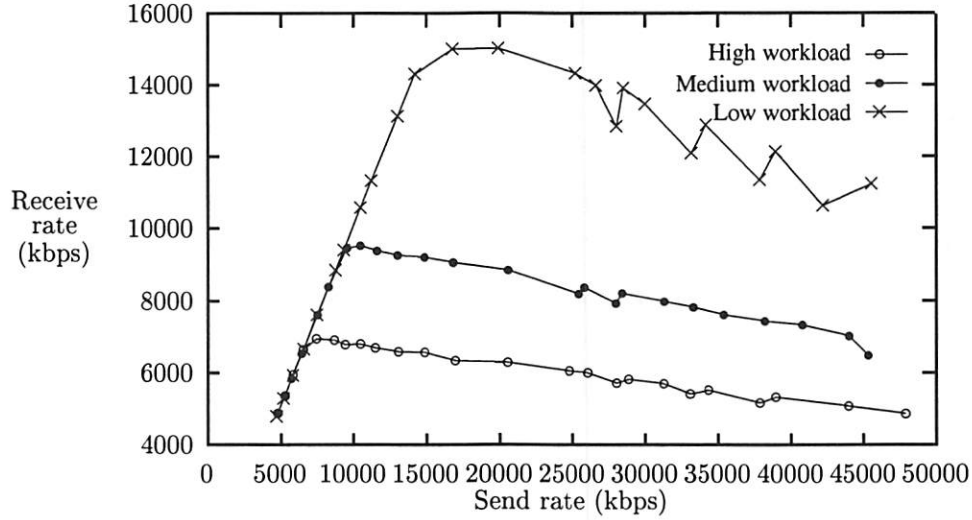


Figure 7: The effects of workload on throughput with the purely interrupt-driven system. Each dot in the graph represents the average throughput of a series of measurements with 2048 bytes UDP packets.

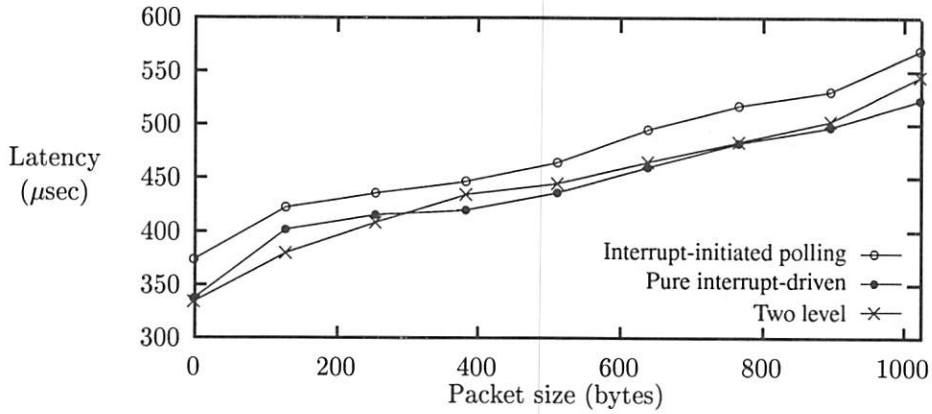


Figure 4: Latency comparison of the three different network handling schemes. Each dot in the graph represents the average latency of a series of measurements with UDP packets.

venting rescheduling of the user thread. When using the two level scheme (see figure 6), the throughput is more stable although there still is some variation in the measurements at high network load. The polling thread may be interrupted by other threads than our test application, and thus small variations may occur.

When using the real-time priority user level threads, we see that the two levels occur at much lower network load (see figure 5). As the real-time threads are not preempted, they are stuck on the same processor unless they are suspended by system calls or they voluntarily suspends execution.

During these tests another problem surfaced—it was often the case that the send rate varied quite a bit, e.g., during tests where a 512 bytes UDP packet was used, the send rate could vary between 32 Mbps and 44 Mbps. This was caused by the sending thread and the interrupt-routine sharing processor. As the load generated by the interrupt-routine on the sending side was lower than 50% polling was not used. In order to evaluate the benefit of polling in this situation, we temporarily reduced the threshold to 20%. This gave a much more stable send rate at approximately 41 Mbps. As the send rate is lower when using a polling thread, this indicates that the polling thread does not run continuously due to the low network load. Thus, the polling routine degrades performance slightly as it consumes processor time without processing interrupts.

5.4 Multithreaded server

In order to evaluate the effects of interrupt-handling and polling on a multithreaded server, we modified the `netperf` program, so that multiple threads were used. One thread receives the packets, and places them in a work queue, and one or more worker threads removes packets from the work queue, and simulates processing of the packet. The amount of work done for each packet can be varied, so that the effects of the workload per packet can be studied. Furthermore, we simulate shared data structures by making it possible to specify that a certain amount of the workload must be performed in a critical region. In the following we first show the results of varying the workload in the case where there is no locking, and then we look at the effects of varying the amount of locking at a fixed workload.

5.4.1 The Effects of Workload

As depicted in figure 1 we would expect—in the case were the purely interrupt-driven system is used—the rate of received data to rise to a maximum level and then drop to a lower level corresponding to the situation where the interrupt-routine has exclusive access to one processor. Using polling this should not occur—instead the rate of received data should stabilize near the maximum level. In figure 7 we see how the average throughput of the purely interrupt-driven system drops as the network load increases. However, there is no indication of the low level of network activity. When using the two level scheme (see figure 8), the throughput is almost the same level during high network load as in the case where there is no overload. Figure 8 also illustrates one of the problems with using a fixed processor load limit for deciding when to use thread-based network handling. When the user thread part of the processing of network traffic is large, the overload situation may arise even though the network traffic is not at or near the

back.

Interrupt rate By monitoring the interrupt rate of a device, an interrupt rate threshold value could be used to decide when the network load is high. The problem is that many device drivers use interrupt batching, i.e., processes multiple packets per interrupt.

Amount of time spent processing interrupts By measuring the percentage of processor time used by the DPC of the device driver, it should also be possible to detect user thread starvation. The measurement of the processing time is complicated by the fact that a DPC may be interrupted, but as the interrupts primarily are hardware related interrupts the impact should be negligible.

We have based our implementation on measuring the interrupt processing time as this is simple to implement. The transition from polled to interrupt-driven I/O is made when the polling thread lacks work to do.

4.3 Operating System Support

As the work done by the polling thread and the interrupt handlers is almost the same, it would be beneficial to integrate support for both interrupt handling and polling in the operating system. By letting a device driver register routines explicitly for polling threads and interrupt handlers, the I/O Manager can take active part in the decisions on what type of I/O handling to use , e.g., by monitor the execution time of interrupt handlers and initiate polling.

5 Results

In the following, we compare our two level scheme with a standard purely interrupt-driven device driver. In order to evaluate the viability of the two level scheme, we look at network latency, thread pinning, and finally we examine the effects of user thread starvation on multithreaded applications.

5.1 Methodology

To produce an overload situation on the receiving machine, a Dual Pentium Pro 200 MHz host was used as the transmitting side, and a Dual Pentium 133 MHz machine as the receiver. Both machines were running Windows NT 4.0 with service pack 3. The two machines were each equipped with two Olicom RapidFire 155 Mbps ATM adapters. All performance tests have been made using the TCP/IP protocol stack shipped with Windows NT on top of the Olicom driver configured to use Classical IP with a PVC between each pair of network adapters. As network load generator we use the network performance measurement tool `netperf`[2]. Again, to overload the receiver, we use the UDP protocol. In our two level scheme we used a threshold of 50%, i.e., the transition to polling was made, if an interrupt-routine used more than 50% of the processing time on a single processor for a period of more than two seconds.

5.2 Latency

To compare the overhead introduced by using the polling routine, we compare the latency of a pure interrupt-driven system, our two level scheme, and an interrupt-initiated polling routine. The interrupt-initiated polling routine is obtained by modifying the two level scheme implementation, so that all the DPC does is to signal the polling thread. The measurements are illustrated in figure 4, and show that the two level and pure interrupt-driven schemes have about the same latency, which is between 25 and 50 μsec higher than the interrupt-initiated polling scheme. Thus, the overhead of monitoring the execution time of the interrupt-routine is negligible, and low latency is achieved at low network load.

5.3 Thread Pinning

When thread pinning occurs, we expect to see two different levels on the rate of received data, one high level corresponding to the case where the user thread and interrupt-routine execute on different processors, and a low level in the case where they are executing on the same processor. We look at user threads running both at normal and real-time priority. User threads at real-time priority should be more vulnerable to interrupt-routines stealing cycles, as they can only be preempted by threads with higher real-time priority.

In figure 5 we show how thread pinning occurs on the receiving machine during a throughput measurement using 1024 bytes UDP packets at various send rates. In order to show the instability and variation in throughput, we conducted 10 measurements for a series of send rates for each priority, and show each of these measurements as a single dot in the graph. As can be seen from the figure, the received rate of the normal priority threads only split into two levels during extremely heavy load. This is due to the fact that only during maximum network load does the interrupt-routine run continuously and thus pre-

[2]Netperf can be obtained from The Public Netperf Homepage at http://www.cup.hp.com/netperf/NetperfPage.html.

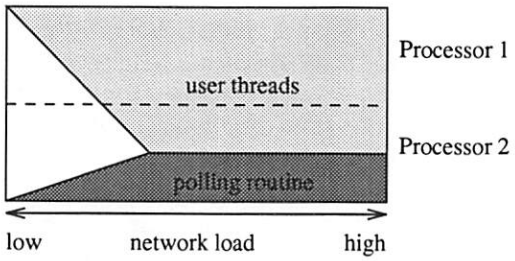


Figure 3: Processor usage as a function of network load on a two processor machine in the case where the use of a polling thread prevents performance degradation.

able to solve the problems regarding thread pinning, interrupt-routines stealing processor time from user threads and locking. This is illustrated in figure 3. Here the polling routine does not steal cycles from the user thread, and thus the processor time is used in a way that maximizes throughput. Furthermore, as there are no interrupts, threads holding a lock cannot be interrupted.

3 Our Two Level Scheme

The problems described in the previous sections lead us to abandon pure interrupt-driven network interface handling. The network handling based on interrupt-initiated threads seems attractive when the network load is high, but it would be nice to avoid the delay caused by both issuing an interrupt and making a context switch in order to process a packet when the network load is low. We therefore use a two level scheme where interrupt-driven servicing of the network interfaces is used until a certain level of network traffic, and above that level, a polling thread scheme is used.

4 Windows NT Implementation

This section provides a closer look at how we have implemented this scheme in Windows NT. First we give a brief description of the relevant parts of Windows NT I/O management as this provides the basis for the further discussion. Then we look at how to detect livelock in Windows NT, and finally we discuss how support for this scheme could be integrated with the current Windows NT I/O subsystem.

4.1 Windows NT I/O Management

In our description, we use a simplified model of a network protocol stack, where we have a transport driver placed on top of a device driver. In Windows NT, the layers interact by passing I/O Request Packets (IRPs) from one layer to the other. This is done via an I/O manager. In the following we describe how data transmission and reception is handled by this model.

On data transmission, the transport driver passes an IRP to the device driver, where the IRP is either processed by a device driver dispatch routine, or—in the case where the device is busy—queued for later processing. When the transmit operation is completed, the IRP is returned to the transport driver. This causes an I/O completion routine to be called in the transport driver. This I/O completion routine is often just a queuing of the IRP for further processing. In the case, where the completion of the transmit operation relies on a hardware interrupt from the device, the dispatch routine would return, and rely on an interrupt handler to complete the transmit operation. The NT interrupt handling consists of two steps - first the hardware interrupt causes the execution of an Interrupt Service Routine (ISR) running at device Interrupt ReQuest Level (IRQL), which does minimal work (e.g., disabling interrupts). This causes a Deferred Procedure Call (DPC) to be queued. This DPC is executed by a software interrupt when the IRQL drops below Dispatch/DPC Level (this is below device IRQL, but above normal thread execution level). This DPC handles the bulk of the processing.

When data is received on a device, the data is passed on to the transport driver in an IRP as described above.

The IRP queues can either be managed by the device driver or the I/O manager. Transmit operations are handled by a set of device driver dispatch routines. These may rely on interrupts to signal the completion of a transmit operation, and thus a part of the transmit handling is placed in the DPC.

4.2 Detecting User Thread Starvation

The main problem is to detect user thread starvation, i.e., when to make the transition between interrupt-driven and polled I/O. We consider the following possibilities:

Length of network data queues By monitoring the length of the network data queues (possibly IRP queues) that are emptied by the user thread, it should be possible to detect when user thread starvation occurs. The problem is that these queues are often internal to the transport driver requiring that the device driver has access to information about the size of the queues in the transport driver. As a transport driver may be bound to several devices, it should only be the IRPs belonging to the device that are reported

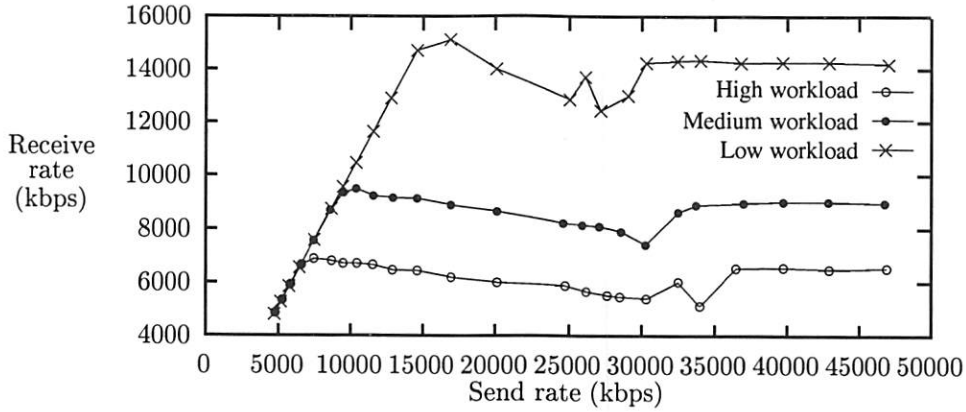


Figure 8: The effects of workload on throughput with the two level system. Each dot in the graph represents the average throughput of a series of measurements with 2048 bytes UDP packets.

peak rate. In our case, where the shift occurs at 50%, which in this case corresponds to a transmit rate of 30 to 35 Mbps, the effects of thread-based network handling are first seen above this level.

5.4.2 The Effects of Locking

In order to evaluate the effects of sharing data, we conducted a series of throughput measurements, where the user threads were required to hold a lock for either 20%, 40% or 60% of the total packet processing time.

In figure 9 we see the effects of interrupt-driven I/O on the part of the graph below the 30000 Kbps send rate, and the effects of using a polling thread above. In the part of the graph representing measurements from interrupt-driven I/O, the slope of the 60% locking curve is considerable steeper than the 20% locking curve, which indicates that increased locking has a negative effect on the received rate. There is no large difference between the curves for 20% and 40% locking, thus the effects of locking are noticeable in the slope of the curves only when a large amount of locking is used. Furthermore, the difference in throughput between the different locking percentages are much lower when the polling thread is used. This indicates that interrupts have influence on locking—especially when intensive locking is used, and that polling threads perform better.

5.5 Summary

Our results show, that problems regarding scheduling of interrupt-routines on multiprocessors exist. In case of extreme network load, user threads may experience thread pinning. Windows NT real-time threads are particularly vulnerable to user thread starvation as they are not rescheduled during high network load due to the lack of preemption. This can be avoided by using polling threads. Alternatively, an interrupt system, which interrupts the thread with the lowest priority, could be used.

When using multithreaded applications on multiprocessors, problems with thread starvation may also be experienced. In the case where user threads are sharing data protected by locks, the effects of the user thread starvation caused by interrupt-routines may get worse. This can also be alleviated by using polling threads.

The fixed threshold used to shift to polling is a problem in the case where applications have different or variable workloads per packet. As workload intensive applications experience starvation at lower processor loads a threshold based on a fixed processor load is not flexible enough to accommodate a wide range of applications. It would possibly be better to base the shift from interrupt-driven to polling on packet discards in the I/O system.

6 Related Work

The problems with livelock and the use of polling as a mean to avoid this was first described by Mogul and Ramakrishnan [3]. Furthermore, they reduced the latency increase imposed by pure polling by using interrupts to initiate the polling thread.

A periodic polling scheme has been suggested by Smith and Traw [6], where clocked interrupts are used to service an ATM network interface in order to reduce the total number of interrupts.

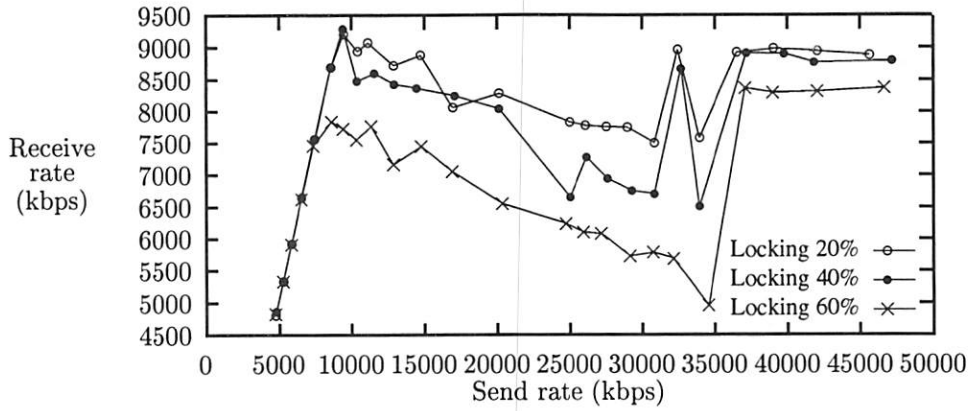


Figure 9: The effects of locking on throughput with constant workload. Each dot in the graph represents the average throughput of a series of measurements with 2048 bytes UDP packets.

In the Windows NT Kernel-Mode Driver Reference Guide [2] it is suggested, that periodic polling should be used to complete sends, when the total load on the processor, on which the interrupt-routine is executing, exceeds a certain level. This would also reduce user thread starvation, but does not address the problems of unevenly balanced load on multiprocessors.

7 Conclusion

We propose a two level device servicing system that uses interrupt handling during low network loads in order to provide low latency and polling during high network loads in order to prevent user thread starvation. This can be integrated in the I/O system of Windows NT.

Our current prototype on a couple of dual processor workstations running Windows NT 4.0 shows that the scheme is able to improve performance on network saturated multiprocessors.

Acknowledgments

We wish to thank Olicom A/S, in particular Tomasz Goldman and Kim R. Pedersen, for providing the necessary ATM hardware, for giving us access to their driver source codes, and for providing support in general. The project is funded in part by The Danish Natural Science Foundation and The Danish National Centre for IT Research.

References

[1] Mats Björkman and Per Gunningberg. Locking Effects in Multiprocessor Implementations of Protocols. In *Proceedings of SIGCOMM '93*, pages 74–83, September 1993.

[2] Kernel-Mode Drivers Reference Guide. Part of the Windows NT 4.0 Device Driver Kit, 1996.

[3] Jeffrey Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[4] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of OSDI '94*, November 1994.

[5] Gerald W. Neufeld, Mabo Robert Ito, Murray Goldberg, Mark J. McCutcheon, and Stuart Ritchie. Parallel Host Interface for an ATM Network. *IEEE Network*, pages 24–34, July 1993.

[6] Jonathan M. Smith and C. Brendan S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, pages 44–52, July 1993.

Coordinated Thread Scheduling for Workstation Clusters Under Windows NT

Matt Buchanan and Andrew A. Chien
Concurrent Systems Architecture Group
Department of Computer Science, University of Illinois
({mbbuchan,achien}@cs.uiuc.edu)

Abstract

Coordinated thread scheduling is a critical factor in achieving good performance for tightly-coupled parallel jobs on workstation clusters. We are building a coordinated scheduling system that coexists with the Windows NT scheduler which both provides coordinated scheduling and can generalize to provide a wide range of resource abstractions. We describe the basic approach, called "demand-based coscheduling", and implementation in the context of Windows NT. We report preliminary performance data characterizing the effectiveness of our approach and describe benefits and limitations of our approach.

1. Introduction

Coordinated scheduling for parallel jobs across the nodes of a multiprocessor is well-known to produce benefits in both system and individual job efficiency [1, 5, 6]. Without coordinated scheduling, parallel jobs suffer high communication latencies between constituent threads due to context switching. This effect is exacerbated if the thread scheduling for individual nodes is done by independent timesharing schedulers. With high performance networks that achieve latencies in the range of tens of microseconds, the scheduling and context switching latency can increase communication latency by several orders of magnitude. For example, under Windows NT, CPU quanta vary from 20 ms to 120 ms [3], implying that uncoordinated scheduling can cause best-case latencies on the order of 10 μs to explode by three to four orders of magnitude, nullifying many benefits of fast communication. While multiprocessor systems typically address these problems with a mix of batch, gang, and timesharing scheduling (based on kernel scheduler changes), the problem is more difficult for workstation clusters in which stock operating systems kernels must be run. Coordinated scheduling reduces communication latencies by coscheduling threads that are communicating, thereby eliminating the context switch and scheduling latencies from the communication latency. Another important problem is that uncoordinated scheduling can reduce system efficiency as increased latency can decrease the efficiency of resource utilization. As high-performance networks and CPUs become more affordable for high-end computing, scheduling emerges as an important factor in overall system performance.

A low-latency, high-bandwidth messaging layer works to bridge the mix of interconnect, operating system, and user applications, delivering the raw performance of the interconnect to the software [14,2,8,9]. Such layers are essential because they make available the high performance of the underlying network hardware to applications. Illinois Fast Messages (FM) is such a messaging layer [14], and is a key part of the Concurrent System Architecture Group's High Performance Virtual Machines (HPVM) project [15], which seeks to leverage clusters of commodity workstations running Windows NT to run high-performance parallel and distributed applications. Fast Messages can deliver user-space to user-space communication latencies as low as 8 μs and overheads of a few μs. However, such levels of performance implies avoiding interrupts and system calls, so systems such as FM use polling to detect communication events, and therefore only delivers peak communication performane when effective coscheduling is achieved.

Coscheduling for clusters is a challenging problem because it must reconcile the demands of parallel and local computations, balancing parallel efficiency against local interactive response. Ideally a coscheduling system would provide the efficiency of a batch-scheduled system for parallel jobs and a private timesharing system for interactive users. In reality, the situation is much more complex, as we expect some parallel jobs to be interactive. Furthermore, in a cluster environment, kernel replacement is difficult at best, so we restrict ourselves to ap-

proaches that involve augmentation of existing operating system infrastructure.

Our approach to coordinated scheduling is Demand-based Coscheduling (DCS) [4, 7] which achieves coordination by observing the communication between threads. The essence of this approach is the observation that only those threads which are communicating need be coscheduled, and this admits a bottom-up, emergent scheduling approach. This approach can achieve coscheduling without changes to the core operating system scheduler and without changes to the applications programs. DCS causes the Windows NT scheduler to schedule communicating threads in a parallel job to run at roughly the same time, thereby minimizing communication latency.

Our implementation of DCS in Windows NT coexists with release binaries of the operating system require a customized device driver for the network card (in this case the Myrinet [13] card). This driver memory maps the network device into the user address space. The device driver, combined with special Myrinet firmware, influences the operating system scheduler's decisions by boosting thread priorities, based on communication traffic and system thread scheduling. The DCS algorithms are designed to drive the node OS schedulers into synchrony, achieving coscheduling among parallel threads that are closely communicating while simultaneously preserving fairness of CPU allocation. Our experiments evaluate an implementation of DCS for Windows NT, demonstrating that such an architecture is feasible, and validating DCS as a –promising approach for coscheduling. However, more extensive experiments are required before stronger conclusions can be drawn.

The remainder of this paper is organized as follows. Section 2 describes demand-based coscheduling briefly and our implementation approach for DCS. Section 3 describes performance with DCS, and section 4 presents some concluding remarks.

2. Demand-based Coscheduling

2.1. Overview

Demand-based coscheduling makes one central observation about the problem of scheduling parallel jobs, that only *communicating* threads need to be coscheduled to overcome scheduling and context switch latencies. DCS is driven directly by message arrivals: Whenever a message arrives, an opportunity for coscheduling arises. If no thread that can receive the message is currently running, DCS decides whether to preempt the current thread to synchronize the sending and receiving threads. The decision may be based on many factors, but in general DCS attempts to strike a balance between maximizing coscheduling performance and ensuring that the CPU is allocated fairly.

At the highest level, DCS has three key elements:

1. Monitoring communication and thread scheduling,

2. Deciding whether to preempt the currently running thread, and

3. Inducing the scheduler to select a particular thread.

The most direct approach to all three elements of DCS is to modify thread scheduler, embedded at the heart of the operating system kernel. However, because our goal is to support clusters running retail operating systems, such an approach has at least three drawbacks. First, modified kernels are unlikely to be run on a large number of systems, so such an approach will preclude large scale use of the coscheduling technology as middleware. Second, kernel modifications will tie the implementation to a particular operating system and version, increasing the software maintenance overhead, and making it difficult to leverage new operating systems features. The third and final drawback is proprietary concerns relating to source and binary distribution. An external implementation is generally more easily distributable.

DCS has been simulated extensively and implemented in the context of Solaris 2.4 [7,4]. The Solaris 2.4 implementation served as a model for and is similar in many ways to our Windows NT implementation

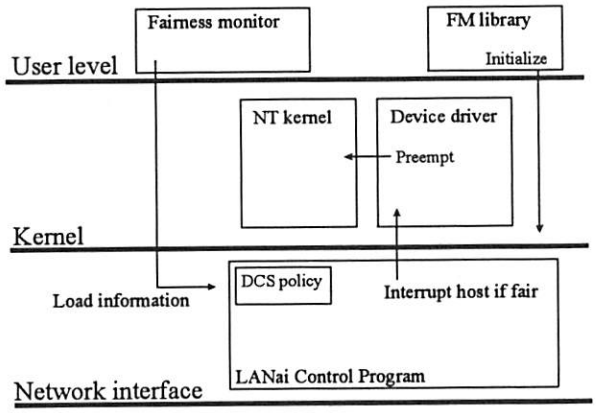


Figure 1. DCS organizational chart

2.2. Implementation

We implemented DCS for Windows NT 4.0 for the Intel *x*86 family of CPUs. Our implementation for Windows NT consists of three distinct parts: a DCS-aware Myrinet driver, the Fast Messages user-level library, a Fairness monitor, and DCS-aware FM firmware (a "LANai Control Program", or LCP) that runs on Myrinet card. These elements interact as shown in Figure 1. The Fairness monitor in combination with the device driver monitors thread scheduling in the system, the modified firmware uses this information to decide whether to preempt the current thread, and the modified device driver induces the kernel scheduler to choose the desired thread for DCS.

2.2.1. Fairness Monitor

Our DCS implementation monitors thread and communication activity to ensure fairness of CPU allocation. The Fairness monitor runs as user-level (and thereby can access the NT kernel's performance data for the length of the run queue). The average run queue length is written to the network card periodically, allowing the Myrinet firmware to moderate the frequency of priority boosts to ensure fair CPU allocations. Status information for the current thread is provided by the device driver as indicated below.

2.2.2. Myrinet Firmware

The DCS aware firmware is a modified version of the FM LANai Control Program and in addition to its basic communication function, the firmware makes preemption decisions based on the monitoring information provided by the Fairness monitor and the device driver.

Based on the run-queue length, current thread information, and the communication activity, the Firmware decides whether the current thread needs to be preempted (via an interrupt) and the device driver invoked to take DCS-related action. The decision procedure used by the Firmware is described below.

To determine whether a given thread is running, the Firmware periodically scans (approximately once per millisecond) the context switch information provided by the device driver. The LCP sets a flag if a communicating thread is running, and when a packet arrives, evaluates the following condition:

```
!threadIsRunning && fairToPreempt()
```

If the condition is true, then the LCP interrupts the host. The fairness criteria is critical and we adopt the approach taken in [4] to decide whether to interrupt the host. For a given thread, we interrupt if the following inequality is true:

$$2^E(T_C - T_P) + C \geq T_q R$$

where

- T_C is the current time,
- T_P is the time the host was last interrupted to schedule this thread,
- T_q is the length of a CPU quantum (120 ms under Windows NT Server [3]),
- R is the number of threads waiting for the CPU,
- E and C are constants chosen to balance fairness and performance.

This approach limits the number of preemptions performed on behalf of a communicating thread by requiring that T_C - T_P, the time since the last preemption, is no less than the time it would take the CPU to service all of the ready threads if each thread ran for its entire quantum. The decay function 2^E and constant C afford some flexibility in tuning the inequality to allow for perturbations, such as those caused by threads that do not expire their quanta and priority-decay scheduling. The Firmware uses the Myrinet card's on-board clock (0.5 μs granularity) to

track T_C and T_P. Under NT Server, the quantum size is constant.

Since the LCP scans the context switch information at one-millisecond intervals rather than for each packet, the information that `fairToPreempt()` uses may be stale. The scanning period involves a tradeoff between per packet overhead and staleness of the data. Since NT typically switches threads on tens of milliseconds time scale, we choose to reduce the per packet overhead.

2.2.3. Device Driver

We explored two basic approaches to the device driver, and describe both here as an illustration of what turned out to be difficult about manipulating the kernel scheduler to achieve the desired schedule. We term these two approaches **thread select** which makes use of a thread select callback, and **priority boost** which uses the thread priority boosting mechanism.

Thread Select Our initial implementation of DCS used NT's *thread select notify callback*, implemented in multiprocessor versions of the kernel. The scheduler passes the handle of a thread it proposes to select, and the callback returns a boolean value that it uses as a hint in deciding whether the given thread is appropriate to schedule. To cause the scheduler to favor a given thread when the thread has messages pending, the callback would simply reject the scheduler's choices until it proposed the preferred thread.

Unfortunately, the thread select notify callback is not suitable for DCS because its influence on thread scheduling decisions is limited. The scheduler uses several other criteria in addition to the callback's return value in choosing a thread to run, including the time the thread has been waiting for a CPU and its processor affinity [10]. Of course, if a competitor thread has a higher priority than a communicating thread, the scheduler may never propose a communicating thread to the callback. Thus, there is no guarantee that a communicating thread will be offered, much less scheduled at an appropriate time. Thus, using the callback to modify the scheduler's behavior was not a viable implementation for DCS.

Priority Boost This approach boosts the priority of a thread DCS would like to schedule which in general causes the NT scheduler to schedule the desired thread. However, since the Windows NT kernel does not export a well-defined interface to device drivers for modifying the priorities of arbitrary threads, (only for boosting the priority of driver created system threads [10]), we were forced to use an undocumented internal interface for thread priority modification. By using a tool called "NTExport" [11] that uses the symbol information distributed with every build of Windows NT (intended for kernel debugging support) to build an export library for the kernel, we exported the appropriate calls to our driver, enabling thread priority modification. (We hope to find a more portable yet equally effective approach to solve this problem.) When a thread needs to be scheduled, the driver's interrupt handler affects a priority boost for the thread.

In addition, monitoring thread scheduling activity is another key function of the device driver. To provide thread scheduling (current thread) information to the Myrinet Firmware, our device driver exploits a kernel callback on each thread context switch to write the context switch information to the Firmware. Thus, the firmware has precise current thread information.

3. Performance Results

We have implemented DCS for Windows NT 4.0 on a cluster of dual-processor Micron brand Pentium Pro machines running at 200 MHz, each with 64 MB of physical memory. Each machine contains a Myrinet PCI interface connected to a Myrinet switch.

Our experimental methodology was as follows: We ran trials of FM's latency benchmark along with zero, one, two, four, and eight CPU-bound competitor threads, passing one million packets on a round trip between a sender and receiver node. We ran ten runs of this test. Each trial reported a histogram of round-trip times in microseconds. We computed the mean value of each bin for each number of competing threads over the ten trials to get the results we report here.

Preliminary results show that DCS improves performance, but that balancing fairness with performance is a tradeoff. We ran a ping-pong latency microbenchmark and a barrier benchmark on a cluster of six dual-processor 200 MHz Pentium Pro machines. Malfunctions in our LANai development tools prevent us from reporting the results of the barrier series here.

Figure 2 shows the wall-clock time-to-completion for FM's latency test with DCS enabled using E=0 and E=-5 and with DCS disabled. Our testing has indicated that for a large number n of compute-bound competitor threads, say four, the NT Server scheduler is fair; that is, we observe each competing thread to receive $1/n$ of the system. For four or more competitors, Figure 2 shows DCS with E=-5 to exhibit behavior similar to that of the NT scheduler alone. Since the latency test measures the wall-clock round-trip time required for a series of messages (in this case, one million), the time required for the entire test to run is an indicator of the average round-trip latency observed.

Figure 3 shows the distribution of round-trip times for the eight-competitor case. The graph illustrates the large contributions that round-trip times as large as 1 second make to the latency benchmark's time to completion as the number of competitor threads grows. Most of the contribution that the aggressive DCS configuration (E=0) makes to time to completion is clustered on the left side of the graph; non-DCS and the more passive DCS configuration (E=-5) show substantial numbers of round trips that last longer than 2 ms to total time to completion. The average latencies for the non-DCS and passive DCS cases swamp the average latency reported in the aggressive DCS case.

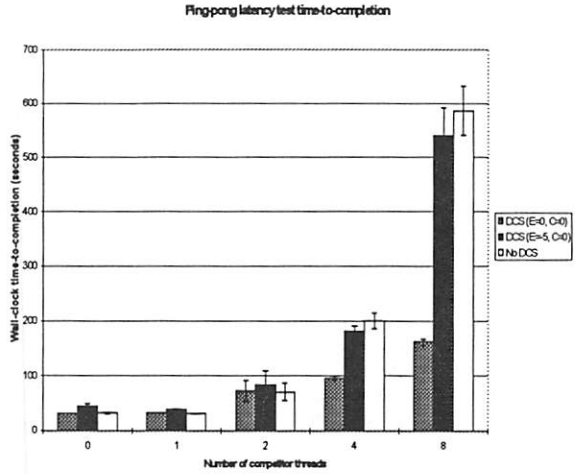


Figure 2. Latency test time-to-completion

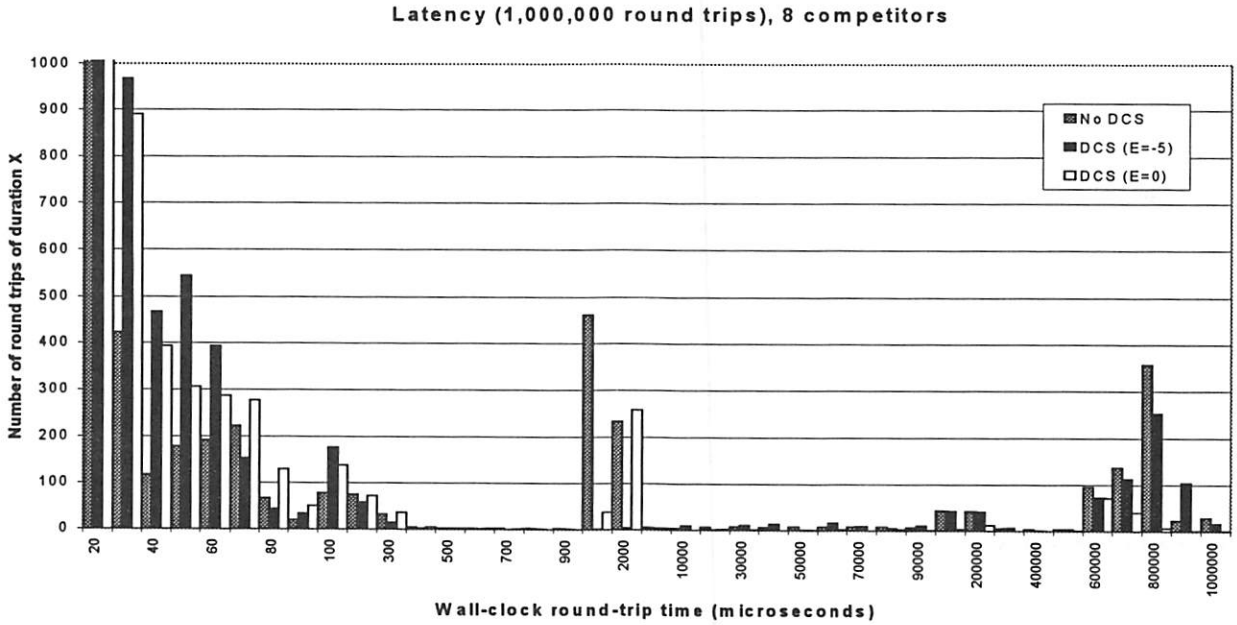


Figure 3. Distribution of latency test round-trip times. For x=20, y=996000 for each configuration.

4. Summary

Our initial performance results are encouraging and suggest that DCS can be implemented and can achieve coscheduling for Windows NT systems. This coscheduling is demonstrated in the improved

performance of our benchmark for communicating threads. Unfortunately, we can only report limited results at this point, but hope to report performance data from a broader array of experiments in the near future.

5. Discussion and Future Work

More complete performance measurements using larger applications with our DCS implementation are clearly an important step. Exploration of the parameter space for our DCS fairness equation and techniques for auto-calibration are of interest. In addition, we have added a blocking primitive to the FM interface that we will use to explore the behavior of spin-block synchronization under NT in addition to the current spin-only synchronization, alone and in the presence of DCS. Beyond that, experiments with multiprocessor nodes, proportional share scheduling, and scheduling a broader array of cluster resources are all challenging directions.

Our experience with external customization of the Windows NT scheduler has mixed results. While we initially believed that the wealth of callbacks and external hooks for NT would make external customization easier, our experience was much less encouraging. The callbacks for thread scheduling were inadequate, and only available in the multiprocessor released kernel. For research such as we have discussed to proceed without NT kernel modificaitons, general, better external access to NT's policies (and mechanisms) must be achieved. Priority boosts are a crude *mechanism* for achieving coscheduling, but an effective callback would influence the scheduler's *policy*, possibly achieving the longer-term scheduler synchrony across the cluster that is our goal. A less ambitious approach would involve simply better access to mechanisms for thread priority modification, obviating the need for recourse to tools like NTExport.

More information

More information is available on our WWW site at http://www-csag.cs.uiuc.edu.

Acknowledgments

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grants MIP-92-23732, NASA grant NAG 1-613. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Microsoft is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

6. References

[1] Ousterhout, J. K. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22-30, October 1982.

[2] Von Eicken, T, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[3] Russinovich, M. Differences between Windows NT Workstation and Server. Available from `http://www.ntinternals.com/tune.txt`.

[4] Sobalvarro, P. G. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1997.

[5] Feitelson, D. G. and L. Rudolph. Coscheduling based on run-time identification of activity working sets. In *International Journal of parallel Programming*, Vol. 23, No. 2, pages 135-160, April 1995.

[6] Dusseau, A. C., R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, 1996.

[7] Sobalvarro, P. G. and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the Parallel Job Scheduling Workshop at IPPS '95*, 1995. Available in Springer-Verlag Lecture Notes in Computer Science, Vol. 949.

[8] Von Eicken, T., A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of*

the 15th ACM Symposium on Operating Systems Principles, December 1995..

[9] Tezuka, H. A. Hori, and Y. Ishikawa. Design and implementation of PM: a communication library for workstation clusters. In *JSPP*, 1996.

[10] Microsoft. Windows NT device driver kit documentation.

[11] Russinovich, M. and B. Cogswell. NTExport documentation. Available from `http://www.ntinternals.com`..

[12] Custer, H. Inside Windows NT. Microsoft Press (Redmond, WA), 1993.

[13] Boden, N., *et. al.* Myrinet—a gigabit-per-second local-area network. In *IEEE Micro*, pages 29-36, February 1995.

[14] Pakin, S., Karamcheti, V. and Chien, A. A. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPP's, *IEEE Concurrency* 5(2), April 1997, pages 60-73.

[15] Chien, A., *et. al.* High Performance Virtual Machines (HPVM): Clusters with Supercomputing Performance and API's, Proceedings of the Eighth SIAM Conference on Parallel Processing, March 1997, Minneapolis, Minnesota.

Creating User-Mode Device Drivers with a Proxy

Galen C. Hunt
gchunt@cs.rochester.edu

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

Abstract

Writing Windows NT device drivers can be a daunting task. Device drivers must be fully re-entrant, must use only limited resources and must be created with special development environments. Executing device drivers in user-mode offers significant coding advantages. User-mode device drivers have access to all user-mode libraries and applications. They can be developed using standard development tools and debugged on a single machine. Using the Proxy Driver to retrieve I/O requests from the kernel, user-mode drivers can export full device services to the kernel and applications. User-mode device drivers offer enormous flexibility for emulating devices and experimenting with new file systems. Experimental results show that in many cases, the overhead of moving to user-mode for processing I/O can be masked by the inherent costs of accessing physical devices.

1. Introduction

The creation of device drivers is one of the most difficult challenges facing Windows NT developers. Device drivers are generally written with development environments and debuggers that differ from those used to create other NT programs. Perhaps most challenging to many developers, NT device drivers must be fully re-entrant and must not block.

NT file systems, a special class of NT device drivers, are particularly complex because the developer must anticipate interconnections between the NT Cache Manager, the NT Memory Manager and the file system.

Writing device drivers is complicated because drivers operate as kernel-mode programs. Device drivers have limited access to other OS services and must be conscious of kernel paging demands. Debugging kernel-mode device drivers normally requires two machines: one for the driver and another for the debugger.

Device drivers run in kernel mode to optimize system performance. Kernel-mode device drivers have full access to hardware resources and the data of user-mode programs. From kernel mode, device drivers can exploit operations such as DMA and page sharing to transfer data directly into application address spaces. Device drivers are placed in the kernel to minimize the number of times the CPU must cross the user/kernel boundary.

User-mode device drivers offer significant development advantages over kernel-mode drivers with some loss in performance and direct access to hardware. User-mode drivers need not be re-entrant. They have full access to user-mode libraries and applications. User-mode drivers can be created with standard development environments, including high-level languages such as Java or Visual Basic. User-mode drivers can be tested and debugged on a single machine without special tools.

User-mode device drivers are especially useful for emulating non-existent devices or implementing experimental systems. They offer significant advantages in cases where their additional overhead can be masked by I/O latency. Particularly for file system development, user-mode device drivers give the developer great flexibility for developing and designing new systems.

The next section describes the Proxy Driver system for creating user-mode device drivers. Section 3 contains descriptions of several sample user-mode device drivers. Section 4 shows that for typical scenarios user-mode device drivers offer performance similar to kernel-mode drivers. Section 5 lists related work, and the last section concludes with a summary and suggestions for future work.

2. The Proxy Driver

Windows NT I/O is packet driven. Upon entering the NT executive, individual I/O requests are encoded in an I/O Request Packet (IRP). IRPs often pass through multiple drivers; winding, for example, from the executive to a file system driver to a hard disk driver and back. IRPs are processed asynchronously.

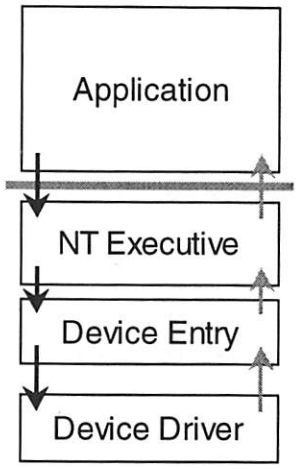


Figure 1, Composition of a Kernel-Mode Driver.

Execution enters a driver through a device entry that represents a logical device. Figure 1 shows the composition of a kernel-mode device driver. I/O requests from the application are converted to IRPs in the NT Executive and passed to the corresponding driver through the device entry.

We implement user-mode drivers using a kernel-mode *Proxy Driver*, see Figure 2. The Proxy Driver acts as a kernel agent for user-mode device drivers. User-mode device drivers connect to the Proxy Driver using a device `open` request (through the File System API) to a special device entry called the *host entry*. The host entry is the doorway to the Proxy Driver's API. The user-mode driver registers with Proxy Driver, informing it of the I/O requests it would like to process. In response, the Proxy Driver creates a new entry in the device table for the user-mode driver called a *stub entry*. The kernel and other device drivers access the user-mode driver through its stub entry. Hidden behind the stub entry, the user-mode device appears as a kernel-mode device. The user-mode driver communicates with the Proxy Driver via `read` and `write` operations on the host device entry.

When an application or the kernel makes an I/O request relevant to the user-mode driver, the IRP is routed through the kernel to the Proxy Driver using the stub entry. The Proxy Driver marshals the IRP and forwards it to the user-mode driver through the host entry. The user-mode driver first processes the request then returns the response to the Proxy Driver through the host entry. The Proxy Driver returns the completed IRP to the kernel through the stub entry.

To simplify user-mode driver development even further, drivers are implemented as Component Object Model

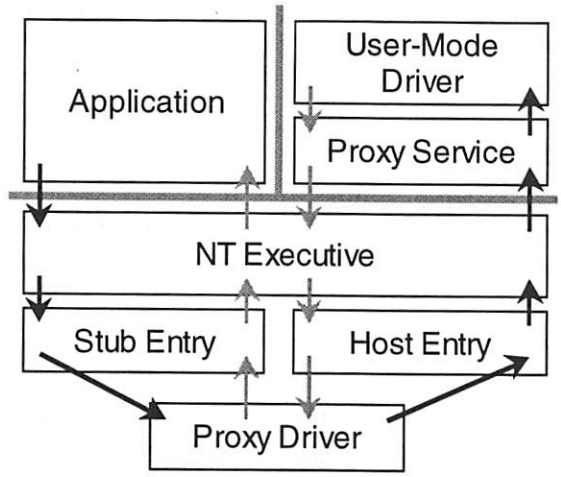


Figure 2, Composition of a User-Mode Driver. The kernel-mode proxy driver passes IRPs to the User-Mode driver through a host device entry and a COM service.

(COM) components. As shown in Figure 2, the Proxy Service sits between the NT Executive and the user-mode device driver. All user-mode device drivers share a single instantiation of the Proxy Driver and Proxy Service. The Proxy Service converts incoming IRPs to calls on the COM interfaces of user-mode drivers. User-mode drivers export COM interfaces for I/O requests they support. Drivers can even inherit functionality from other drivers through COM aggregation. Appendix A lists the IDL definition for `IDeviceFileSink`, the interface used to deliver the most common file IRPs.

The Proxy Service operates under control of the Windows NT Service Manager [4]. It can be stopped or started through the NT Service control panel. At startup, the Proxy Service dynamically loads the Proxy Driver into the kernel. It then looks for user-mode device driver components in the system registry under the key `HKEY_LOCAL_MACHINE\SOFTWARE-\URCS\ProxyDevices`. For each registered component, the service opens a handle on the Proxy Driver and creates a stub entry in the NT Executive. Information stored in the system registry determines where the user-mode device will appear in the Windows NT I/O namespace. System administrators control access to the Proxy Driver by restricting access to the `ProxyDevices` registry key.

The Proxy Driver does not support the NT Fast I/O path or filter drivers. The Fast I/O path is a mechanism by which the NT Executive passes non-blocking I/O requests directly to device drivers without creating an

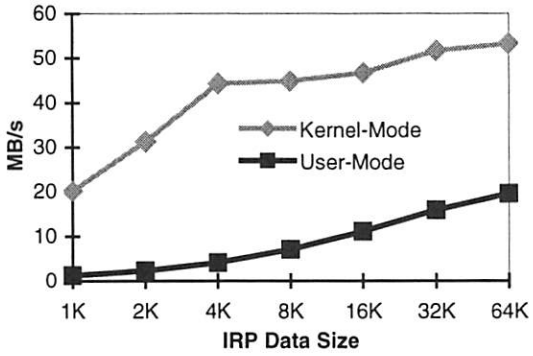


Figure 3, Raw Driver Throughput. IRPs are completed with no side effects.

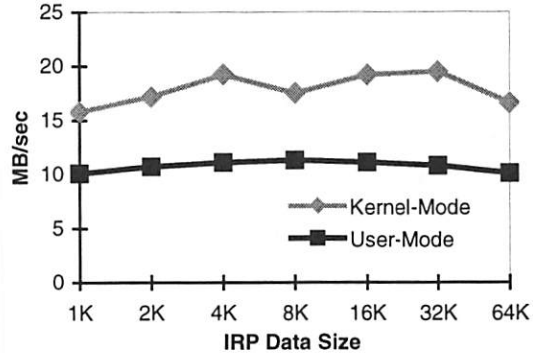


Figure 4, Writing data into either a kernel-mode RAM disk or a user-mode virtual memory disk.

IRP. Filter drivers are drivers that sit between the NT executive and another driver.

The user-mode driver lives in an ideal world. It receives only IRPs destined for its device entry, the stub entry. Because the user-mode driver runs in user-mode, it has access to all traditional user-mode APIs and resources. Finally, because it receives IRPs from the Proxy Driver only after a `read`, the user-mode driver can execute sequentially as a single-threaded program; it need not be re-entrant. Sophisticated user-mode drivers can handle multiple concurrent requests either through asynchronous I/O calls on the host entry or via multiple threads of execution. User-mode driver programmers can choose a level of sophistication appropriate for their domain.

3. Example Drivers

We have created a number of example user-mode drivers. Each is a COM component.

- `rawdev`: A null device for testing user-mode driver performance, `rawdev` completes all IRPs successfully with no side effects.
- `vmdisk`: Similar to a RAM disk, the Virtual-Memory Disk uses a region of virtual memory to emulate a physical disk.
- `efs`: The Echo File System acts as a proxy for another file system. It converts incoming IRPs to Win32 File API calls on the "echoed" file system.
- `ftpfs`: The FTP File System mounts a remote FTP server as a local file system. Incoming IRPs are converted to outgoing FTP requests using the WinInet APIs.

4. Performance

User-mode device drivers trade performance for code simplicity. Whereas a request on a kernel-mode driver would cross the user/kernel boundary twice, each request on a user-mode driver must cross the user/kernel boundary at least four times. In this section we present performance results gathered from the Proxy Driver implementation on Windows NT 4.0 Workstation using a 133MHz Pentium processor.

In Figure 3, raw driver throughput is shown for two functionally equivalent device drivers: one a kernel-mode driver and the other a user-mode driver using the Proxy Driver. Each driver reads, but doesn't copy, the data in an incoming IRP. The great disparity between performance for the two drivers is a result of two factors. First, requests for the user-mode driver must cross the user/kernel boundary twice as often as requests for the equivalent kernel-mode driver. Second, although kernel-mode drivers can directly access IRP data, data bound for user-mode drivers must be copied from the stub IRP to the host IRP. As would be expected, total throughput increases with larger packets as the cost of crossing the user/kernel boundary is amortized. Kernel-mode throughput decreases below 4K as sub-page requests require additional alignment processing.

Figure 4 compares the difference in performance for an application writing to a kernel-mode RAM disk device versus a user-mode virtual memory disk device. Performance for the user-mode driver is better than that in Figure 3 for small requests due to aggregation by the NT Cache Manager. Overall performance for the kernel-mode driver decreases from Figure 3 because the driver must now copy data from the IRP to the RAM

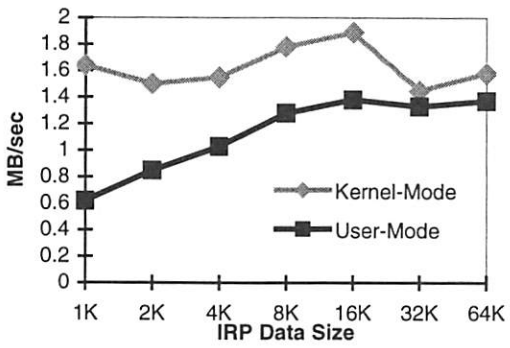


Figure 5, File system throughput.

disk. Note that the Proxy Driver was already making a copy from the kernel's address space into that of the driver.

The relative performance of file system drivers is shown in Figure 5. The kernel-mode file system is Microsoft's NTFS on a physical disk. The user-mode file system, `efs`, forwards I/O requests to NTFS using the Win32 file APIs. Total system throughput is reduced primarily as a result of accessing the physical disk. The difference in performance between the two drivers is essentially the cost of using a user-mode device driver.

Figure 5 demonstrates that particularly when using an external device, such as a disk or network, the impact of using a user-mode driver is minimal. By exploiting full access to user-mode resources, user-mode device drivers can achieve better performance than kernel-mode drivers. The `ftpfs` driver, for example, shares the WinInet cache with user-mode applications, such as Internet Explorer, to reduce network traffic and optimize performance.

5. Related Work

Other researchers have noted the benefits of supporting user-mode device drivers.

Frigate [8] supports user-mode file servers using a dispatch layer in the UCLA Stackable Layers file system [6]. Applications issue file I/O requests either through kernel calls to user-mode servers or directly through a server backdoor. A contemporary of the Proxy Driver, Frigate requires a kernel supporting UCLA Stackable Layers. Frigate has been used to implement an Enigma encryption layer above SunOS file systems.

Bershad and Pinkerton's *watchdogs* [1] are user-mode file-system components. A watchdog is attached to a particular point in the file-system namespace. File I/O requests at the point of the watchdog are intercepted. The watchdog may refuse the I/O operation, perform it on behalf of the system, or return the request to the underlying file system. Watchdogs are intended to allow unsophisticated users to modify the behavior of isolated file system functions. Bershad and Pinkerton use a special communication channel to pass I/O requests to watchdogs.

The HURD system [2] supports the concept of user-mode file systems via a mechanism called *translators*. All files requests are sent through Mach ports. User file systems are just providers of "file" ports. Sample systems proposed include a file system level FTP client and a `/proc` file system similar to [7].

Unlike watchdogs and the HURD system, the Proxy Driver makes no modifications to the NT kernel. It exploits the NT I/O architecture, particularly IRPs, to provide simple, efficient user-mode device drivers for either file system or device operations.

The Semantic File System (SFS) [5] is a user-mode file system that creates dynamic directories based on indexed search criteria. For example, the SFS `/subject:report` directory contains one file for each email message with the word "report" in the subject line. Client machines connect to an SFS server using the NFS protocol.

Similar in principle to our `ftpfs`, the Alex file system [3] provides access to FTP archives using the NFS protocol. NFS requests on the Alex server are converted to FTP requests and forwarded to the appropriate server.

SNFS [9] is a generalized NFS server that supports an internal Scheme interpreter. File requests on the NFS server are processed through user-loaded modules written in a version of Scheme. Proposed modules include union file systems, copy-on-write file systems, and FTP and HTTP file systems.

The Proxy Driver is more efficient than NFS-based user-mode file systems. By exporting native IRPs, the Proxy Driver avoids costly re-marshaling to a foreign I/O model such as NFS.

6. Conclusions

We have described our Proxy Driver system for creating user-mode device drivers. The Proxy Driver resides in the kernel and passes I/O requests to user-mode drivers through a host device entry. User-mode drivers are much easier to write and debug than kernel-mode drivers. Although limited in scope to drivers

needing no kernel-mode hardware access, user-mode drivers offer the programmer great flexibility. User-mode device drivers are particularly effective in cases where physical I/O dominates driver computation.

We are currently developing a toolkit for creating user-mode file system drivers. The toolkit will provide COM components for volatile and persistent cache management, name-space manipulation, and file system layering. With the toolkit, developers will be able to create simple file watchdogs or fully functional file systems in as little as a few hundred lines of code.

Acknowledgments

During the development of the Proxy Driver, the author was supported by a research fellowship from Microsoft Corporation.

Bibliography

[1] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *Computing Systems*, Spring, 1988.

[2] M. Bushnell. Towards a New Strategy of OS Design. In *GNU's Bulletin*, January 1994.

[3] V. Cate. Alex- a Global Filesystem. In *Proceedings of the USENIX File System Workshop*, pp. 1-11. Ann Arbor, MI, May 1992.

[4] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[5] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. OToole, Jr. Semantic File Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 16-25. Pacific Grove, CA, October 1991.

[6] J. S. Heidemann and G. J. Popek. File-System Development with Stackable Layers. In *ACM Transactions on Computer Systems*, vol. 12(1), pp. 58-89, 1994.

[7] T. J. Killian. Processes as Files. In *Proceedings of the USENIX Software Tools Users Group Summer Conference*, pp. 203-207, 12-15 June 1984.

[8] T. H. Kim and G. J. Popek. Frigate: An Object-Oriented File System for Ordinary Users. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pp. 115-129. Portland, OR, June 1997.

[9] T. Lord. Extensible Linux NFS server soon to be available. In *comp.os.linux.announce*, September 1996.

Appendix A

The `IDeviceFileSink` interface is used to deliver common file IRPs to user-mode drivers.

```
interface IDeviceFileSink : IUnknown
{
   HRESULT Create(
        [in] IDevIrp *pIrp,
        [in] IDevSecurityContext *pCtxt,
        ULONG Disposition,
        ULONG Options,
        ULONG FileAttributes,
        ULONG ShareAccess,
        ULONG EaLength,
        LARGE_INTEGER AllocationSize);

   HRESULT Cleanup(
        [in] IDevIrp *pIrp);

   HRESULT Close(
        [in] IDevIrp *pIrp);

   HRESULT Shutdown(
        [in] IDevIrp *pIrp);

   HRESULT Read(
        [in] IDevIrp *pIrp,
        LARGE_INTEGER ByteOffset,
        ULONG Length,
        ULONG Key);

   HRESULT Write(
        [in] IDevIrp *pIrp,
        LARGE_INTEGER ByteOffset,
        ULONG Length,
        ULONG Key);

   HRESULT DeviceControl(
        [in] IDevIrp *pIrp,
        ULONG IoControlCode,
        ULONG InputBufferLength,
        ULONG OutputBufferLength);

   HRESULT QueryInformation(
        [in] IDevIrp *pIrp,
        ULONG Length,
        FILE_INFORMATION_CLASS FIClass);

   HRESULT SetInformation(
        [in] IDevIrp *pIrp,
        ULONG Length,
        FILE_INFORMATION_CLASS FIClass,
        [in] IDevFileObject *pFileObj,
        BOOL ReplaceIfExists,
        BOOL AdvanceOnly,
        ULONG ClusterCount,
        ULONG DeleteHandle);

   HRESULT FlushBuffers(
        [in] IDevIrp *pIrp);
};
```

Measuring Windows NT—Possibilities and Limitations

Yasuhiro Endo, Margo I. Seltzer
{yaz,margo}@eecs.harvard.edu
Harvard University

Abstract

The majority of today's computing takes place on interactive systems that use a Graphical User Interface (GUI). Performance of these systems is unique in that "good performance" is a reflection of a user's perception. In this paper, we explain why this unique environment requires a new methodological approach. We describe a measurement/diagnostic tool currently under development and evaluate the feasibility of implementing such a tool under Windows NT. We then present several issues that must be resolved in order for Windows NT to become a popular research platform for this type of work.

1. Introduction

In recent years, computer systems have become increasingly interactive, most often based on Graphical User Interfaces (GUI). In these systems, users interact with the computer far more frequently than in traditional computer systems, such as those based on a command-line interface or systems used for scientific computation or transaction processing. Another factor that makes these interactive systems different from conventional systems is that "performance" is determined by the user's opinion. This metric, *user-perceived performance*, is different from the performance metrics most commonly used, in that it is affected greatly by the subjective judgment and physical limitations of users. In order to measure and improve user-perceived performance, we need a new methodology that takes these factors into account. We are currently developing such a methodology and a set of diagnostic tools to gather data that will allow us to improve user-perceived performance.

This paper explains how interactive systems are different from conventional systems and why we must devise a new measurement methodology to evaluate interactive systems. As Windows NT is one of the most commonly used GUI platforms, we will evaluate the feasibility of conducting such research on Windows NT. We begin by identifying the differences between measuring interactive systems and conventional systems. We summarize our past efforts to measure interactive systems in Section 3. Section 4 presents the motivation for our current measurement project and outlines the design of the measurement/diagnostic tool currently under development. We then evaluate the feasibility of implementing this tool on Windows NT in Section 5. Section 6 outlines the problems that need to be addressed if NT is to become a popular research platform for this type of work. We present our conclusions in Section 7.

2. The challenges of interactive system measurement

Typically, we use benchmark programs to rate system performance. The most commonly used technique is to measure how quickly a system handles a sequence of requests; from this data, we calculate the bandwidth or the throughput of the system. Systems that achieve high scores in benchmarks are thought to have good performance, and scoring high in well-known benchmarks helps sell systems—both in the commercial market and in the research community. This is one of the reasons that a great deal of effort goes into making systems achieve high benchmark scores. We also use benchmark programs to guide us in the optimization process. A good set of benchmark programs helps us optimize the system effectively by identifying performance bottlenecks. However, it is often difficult to devise a good benchmark that stresses the system in a realistic manner. A bad benchmarking methodology is misleading, encouraging us to optimize parts of the system that have little or no impact on the performance visible to users.

The task of benchmarking interactive systems, such as Windows NT, is further complicated by adding a user into the performance equation. It is known that human judgment of performance is based roughly on the response time or the latency with which the system handles each user request. Generally speaking, collecting response time information in a benchmark is more difficult than calculating the throughput. Moreover, determining how different response times affect users' perception is difficult because perception is greatly influenced by factors such as the user's attitude, expectation, and physical limitations [10]. These factors make benchmarking interactive systems extremely difficult and unique. It is clear that we cannot simply apply conventional techniques to measure interactive systems.

Nonetheless, the majority of today's benchmarks use

conventional techniques that are inappropriate for interactive system measurement. First, these benchmarks often rely on throughput-based metrics. Interactive users do not evaluate system performance based on how quickly a system can respond to a sequence of requests, but how quickly the system can respond to each individual request. Throughput-based metrics do not capture how quickly the system was able to handle *each* of the requests nor do they report the variance observed for the different events. Second, it is considered good practice to use traditional statistical techniques to present our data. We strive for stable, statistically significant results and make every effort—such as removing the system from the network and rebooting the system before each trial—to ensure that the benchmark produces stable output. These experiments inaccurately model the environment in which the system is actually used and ignore the most important and interesting situations we can measure—anomalies. We, as users of interactive system ourselves, often experience situations in which an operation takes an unexpectedly long time for no apparent reason. We, as researchers, must work to eliminate these anomalies; they frustrate users and reduce the user-perceived performance of the system. Experts on human-computer interaction have long noted that expectation has a significant effect on user-perceived performance. Users are surprisingly forgiving when they are waiting for operations they expect to take a long time but are unforgiving when an operation that they expect to complete quickly does not.

We have been working continuously to narrow the gap between the techniques in use and an ideal interactive system measurement technique. We will explain what we have done so far, what we have learned, and on what we are currently working. We will then evaluate the feasibility of implementing the proposed system on Windows NT.

3. Previous measurement projects

In an effort to make systems research more relevant to the majority of computer users in the world, we have undertaken a number of projects to help us understand the performance of personal computer operating systems, such as Windows and Windows NT, and the applications that traditionally run on these platforms. In this section, we discuss these projects from the perspective of the benefits derived from using such a widely popular platform and the challenges it imposed.

3.1. The Measured Performance of Personal Computer Operating Systems

With the realization that the research community must understand more about commodity systems, we set out to measure and understand the behavior of commodity systems and how they compare to traditional research operating systems [2]. We measured and compared Windows NT 3.50, Windows for Workgroups 3.11, and NetBSD 1.0. We used the Pentium Performance Counters [4] to obtain performance statistics including execution time, on-chip cache miss counts, and segment register loads. The Pentium Performance Counters are accessible only from the supervisor mode. Accessing the performance counters under NetBSD was straightforward, since we could freely modify the kernel to introduce the code to manipulate the counter. Under Windows for Workgroups, we used the VxD interface, which allows any user program to introduce supervisor-mode code into the kernel and invoke it directly. Under Windows NT, we took advantage of its installable device driver interface. This interface allows a third party to implement and install a device driver into the NT kernel dynamically. Following the documentation provided in the Windows NT Device Driver Kit [8], we implemented and installed a kernel-mode driver that makes the Pentium Performance Counter registers appear as ordinary device files.

The tools we built allowed us easy access to the performance counters, but the lack of Windows and Windows NT source code limited our ability to interpret measurement results and draw useful conclusions. Explaining the results of benchmarks was made difficult by the fact that we could not confirm our suppositions by code analysis. In many cases, we had to write several new benchmarks to explain the results of one benchmark, and in many cases, writing and measuring the additional benchmarks did not help us fully explain the results. The lack of source code access also meant that we could not isolate and measure specific parts of the kernel. This limited our ability to explain and further understand interesting behavior that the systems exhibited.

Perhaps the most important lesson we took away from this project was the realization that throughput metrics do not always correlate with the user-perceived performance. One of the benchmarks in this study measured how quickly the systems executed a script using Wish, a command interpreter for the Tcl language that provides windowing support using the Tk toolkit [5]. The results of this benchmark were greatly affected by the aggressive optimization that NetBSD and the X-Windows system applied to the input stream. When many requests arrive at the server in a short period of time, the system tries to minimize the server-client communication overhead by sending multiple requests per communication round trip. We observed similar behavior from Windows

NT running Microsoft Powerpoint when processing ten page-down requests. The processor performed five to six times more work[1] when the requests were fed into the system at a realistic rate of about 10 characters per second than when all the requests were fed as quickly as possible. These optimizations help systems perform better on throughput metrics, but often have adverse effects on user-perceived performance. This observation led to our next study.

3.2. Using Latency to Evaluate Interactive System Performance

We set out to establish an appropriate set of techniques for measuring interactive system performance [3]. Previous measurement techniques relied almost entirely on throughput-based measures [1][7], ignoring the fact that throughput and user-perceived performance are different in today's popular GUI environments. User-perceived performance might coincide well with throughput in compute-intensive computations such as scientific computation and compilation, but not necessarily with more interactive applications, such as word processors, spreadsheets, and games.

In this study, we measured the performance of interactive systems using the response time that users experience when running commonly-used applications. We recognized that the response time or the latency of the system handling each user-initiated event, such as a keystroke, correlates better with user-perceived performance than does throughput. We designed and implemented techniques to measure event-handling latency in commodity operating systems and applications, and used these techniques to measure Windows NT versions 3.51 and 4.0 and Windows 95. Since we could not instrument real applications, we devised measurement techniques based on two assumptions. The first assumption was that the CPU is idle most of the time and becomes busy only when it is handling an event. The second assumption was that applications are single-threaded and only call the Win32 API `GetMessage()` to block waiting for new events, after they have completed handling all previous events.

The combination of these two techniques allowed us to measure the latency of user-initiated events *when our assumptions were met*. While we were able to measure how long a simple application, such as Notepad, spent handling each keystroke event, we were unsuccessful in measuring complex applications such as Microsoft Word, or more complex system states, such as multiple applications running concurrently. In terms of understanding user-perceived performance, we identified the crucial difference between throughput and user-perceived performance. Common cases dictate throughput performance—the parts of the system in which the most time is spent are ultimately reflected in throughput metrics. User-perceived performance is dictated by how frequently the user is annoyed and the extent to which the user is annoyed by each occurrence. No matter how frequently they occur, events with latencies below the threshold of user perception do not annoy the user and are therefore irrelevant in the user-perceived performance equation. Conversely, if an event takes sufficiently long to be annoying, its contribution to user-perceived performance is far greater than is suggested from its frequency or percent of total execution time.

4. A New Measurement Methodology

Although constructing a performance metric that captures the subtleties of user subjectivity is beyond the scope of our expertise, we can use the lessons learned in the prior studies to measure and improve the performance of interactive systems. Since the user's judgment of performance is subjective and events that annoy the user are important, we must capture the situations that annoy users. By understanding how these problems arise and correcting the system to avoid such situations, we can improve user-perceived performance.

The measurement system we are building monitors the system under normal operation with a real user on the console. This allows us to exercise the target system in a realistic manner. Controlled experiments based on artificial benchmarks yield stable, statistically significant, reproducible results but often fail to be realistic. In these test cases, it is common to disconnect the machine from the network and/or reboot the machine before each experiment so that the various caches in the system are in a known state. Unfortunately, this is not how most people use their systems, so the benchmarks do not accurately reflect actual use. Real systems often run multiple programs and daemons simultaneously and are constantly affected by external events such as packets arriving on the network. Many performance problems are created by these unpredictable interactions.

For our measurement system, we rely on the user to decide when performance is unacceptable. The user of the system notifies the measurement tool immediately—by clicking a button on the screen—after or while experiencing unacceptable performance. The tool then records the latency of the operation that frustrated the user and dumps data describing the last several seconds

1. Inferred from CPU occupancy time.

of system state leading up to the unacceptable behavior; data that allows us to diagnose the cause of the long latency event. The technical challenges are to identify the data we need to collect and determine how best to collect such data without imposing high overhead (either in time or storage). In the next section, we discuss the type of information we have determined necessary to collect and evaluate the possibilities of collecting such information under Windows NT.

5. Collecting data

Using the measurement system outlined above, we hope to be able to find instances in which interactions between processes and daemons, process scheduling policy, or disk scheduling policy is causing irritating delays in the processing of user-initiated events. Based on earlier work, we have determined some of the data that we must collect. In the sections that follow, we describe the data and the techniques for collecting such data in the Windows NT environment.

5.1. Latency of user-initiated events

Since user irritation is caused by slow response time combined with the expectation of fast response time, it is vital that we measure how long the system spends processing each event. We had moderate success collecting this data in our earlier studies, but we relied upon two simplifying assumptions: that there are no background tasks and that users run a single application at a time. We can no longer afford to make such assumptions. Real users often run more than one application concurrently and many applications are multi-threaded and perform background processing. In such an environment, the user and the application are the only parties who know when event-handling begins and ends. Since it is extremely difficult to instrument users, we must rely on applications to provide this information to the measurement system. This can be accomplished in one of three ways. First, the application can keep track of its own event latencies. This is likely to provide the most accurate data, but it is also the least practical in that it requires access to the application source code (not to mention actually modifying all the applications a user is likely to use). Second, we could use interposition to intercept every application call in a measurement library and then deduce the event latencies based on this trace. This process is feasible by substituting our own libraries for the standard DLLs, but interpretation of the trace output is error-prone. The third technique is for the operating system to try to extrapolate event beginning and ending times. This introduces the potential for the greatest margin of error and requires access to the operating system source code. None of these approaches is particularly desirable, and all require source code availability, which makes it troublesome to collect this type of data under Windows NT.

Even if we could capture latencies using one of these techniques, there is a component of response time that cannot be captured by the application alone. The time that an application spends processing a user-initiated event does not include the time that the system spends delivering a hardware event, such as a mouse click, to the application. Although we have yet to measure such a case in our constrained environment, it is likely that, in a real environment with multiple runnable threads, the delivery time could contribute significantly to latency. To measure event-delivery time, we must timestamp every keyboard and mouse interrupt and determine when the system actually delivers the corresponding higher-level event to a particular application. In Windows NT, timestamping keyboard and mouse events is easily accomplished by modifying the keyboard and the mouse drivers. However, associating a keyboard or mouse input to a higher-level event delivered to the application is challenging. Doing so requires instrumentation of multiple points in the path executed as an input event is processed. This cannot be done without OS source code access. While the installable driver interface provides the ability to collect statistics localized to the driver, it fails to provide us a way to examine or modify the other parts of the system.

5.2. Status of threads running in the system

To identify the cause of a performance problem, it is essential to know when the required thread was able to run, when it was not, and why. Windows NT has an extensive performance monitoring interface. Objects in the system such as threads, processes, processors, disks, and network protocols maintain a set of performance statistics that can be retrieved from user-level, as is done by the statlist program [9]. Using this interface, it is possible to determine the state of a thread, whether the thread is blocked, and the reason why it is blocked.

5.3. Queue States

If the performance of a system is limited because a critical thread is blocked, we must eliminate such waits or shorten their duration. One prime example of such a queue is the disk queue. Using the performance measurement interface described in Section 5.2, we can obtain various statistics including the number of read and write requests and the length of the queues. Unfortunately, the NT performance monitoring interface does not reveal the contents of the disk queue, because such information is localized to the disk driver. However, by

replacing the driver, we can identify the contents of the disk queue. Unfortunately, we cannot directly relate a specific device request to the thread that generated it. Determining this information requires that the driver examine thread states in the core of the kernel to which the device driver interface does not allow access.

5.4. Kernel profile

Kernel profiling can provide detailed information about the inner workings of the operating system, revealing where a thread is spending its time inside the kernel. In cases where a major component of the response time is due to the thread executing inside the kernel, profile output can help identify system bottlenecks. For this study, we need to extend conventional profilers, because we are interested in the data from a specific subinterval, namely the interval during which the user was waiting, not the interval during which the profiler was running. Ordinary profilers do not maintain enough information to calculate subinterval profiles. Typically, profiling requires having source code access to the system being profiled. While it might be possible to construct a profiling system using a binary instrumentation tool such as Etch [6], the information produced would be of limited utility without source code with which to analyze the data.

The requirement of the measurement tool described in Section 4 goes beyond the provisions of Windows NT. Table 1 summarizes our needs for constructing an interactive performance monitoring tool and identifies the classes of data for which source code access is required. In most of the cases, some data is available without source code, but source code is required to analyze the data and relate it to appropriate user actions. For example, although the installable device driver interface provides the ability to measure and experiment with some areas of the system, it fails to provide information about the kernel's inner workings. NT's performance monitoring interface also provides a convenient way to monitor parts of the system, but the disconnect between the performance monitoring tools and the device level interfaces limits its utility.

6. Difficulties in conducting research without source code

In this section, we discuss several of the difficulties we experienced conducting research using Windows NT. Each of the following problems is serious enough to make NT an unsuitable platform for some systems research. In order for NT to become a good research platform, both Microsoft and the research community must work together to address these problems. We have found that detailed performance analysis and understanding requires access to kernel (and application!) source code. First, without source, it is often impossible to confirm hypotheses about system behavior. In both of our previous measurement projects, the lack of source code access made it impossible to draw definite conclusions. In some cases, we were able to make definitive statements by taking several additional measurements to confirm our suspicions, but in many cases, we simply could not substantiate our suppositions—all we could do was increase the probability that our intuition was correct by taking additional measurements. While this is the norm in some natural sciences, it is particularly frustrating when we know that the answer is easily verifi-

Data Collected	Methodology	Source	
		Required for data collection	Required for analysis
Per-event latencies	Detailed accounting of threads' CPU activities.	NO	YES
	Instrumenting the application.	YES	YES
Thread status	NT performance monitoring interface.	NO	YES
Queue status	NT performance monitoring interface.	NO	YES
	Modify device drivers.	NO	YES
Kernel profile	Source code profiler.	YES	YES
	Binary instrumentation tool.	NO	YES

Table 1: Data Collection Summary. This table shows how we might collect the different types of data we need and whether we can collect such data in the absence of source code. Notice that in most cases, the utility of the data is diminished by the absence of the source code.

able.

In some cases, the lack of source code prevented us from pinpointing the exact cause of a performance problem. In systems with source code, we could often find the cause of the performance that we observed and, if desired, modify it and remeasure. In the case of NT, the best we could do was offer vague suggestions for alleviating system bottlenecks. The goal of many research projects is to find an exact answer to a question or solve a problem so that the system will perform better. Windows NT does not allow researchers to achieve this goal in many circumstances. This makes conducting research a frustrating experience.

While Microsoft does offer source code licenses for NT, we find that the restrictions are so limiting that they are in direct opposition to University policy. In particular, requiring students to sign non-disclosure or confidentiality agreements is problematic, particularly in the case of undergraduates. Equally problematic is prohibiting a class of students (those who come from certain foreign countries) from participating in projects that use source code. Finally, there is concern over publication. If we have access to source code, and use that access to explain results that we observe, does our confidentiality agreement prohibit us from explaining our results? That is certainly an untenable situation.

7. Conclusion

The unique manner in which user-perceived performance is determined demands that we develop new techniques to measure and improve interactive system performance. Windows NT is an attractive platform for such research. Conducting practical research that can solve the type of problems that millions of people experience every day excites many researchers. We have demonstrated that it is possible to perform interesting research on NT, but we are approaching the limit that a proprietary system places on the type of measurement research that can be conducted. Researchers expect and require more freedom from the platform they use; we need better hooks into the operating system to provide the detailed information we seek or fewer restrictions on source code licensing.

8. References

[1] Business Applications Performance Corporation, SYSmark for Windows NT, Press Release by IDEAS International, Santa Clara, CA, March 1995.

[2] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Mike Smith, "The Measured Performance of Personal Computer Operating Systems," *ACM Transactions on Computer Systems 14*, 1, February 1996, pages 3-40.

[3] Yasuhiro Endo, Zheng Wang, J. Bradley Chen and Margo Seltzer, "Using Latency to Evaluate Interactive System Performance," *Proceedings of the Second Symposium on Operating System Design and Implementation*, October 1996, pages 185–199.

[4] Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual,* Intel Corporation, 1995.

[5] John K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.

[6] Dennis Lee, Ted Romer, Geoff Voelker, Alec Wolman, Wayne Wong, Brad Chen, Brian Bershad, and Hank Levy, Etch Overview, http://www.cs.washington.edu/homes/bershad/etch/index.html.

[7] M. L. VanNamee and B. Catchings, "Reaching New Heights in Benchmark Testing," *PC Magazine*, 13 December 1994, pages 327-332. Further information on the Ziff-Davis benchmarks is available at: http://www.zdnet.com/zdbop.

[8] Microsoft Corporation, *DDK Platform*, Microsoft Developer Network, Redmond, Washington, January 1996.

[9] Microsoft Corporation, *Win32 SDK*, Microsoft Developer Network, Redmond, Washington, January 1996.

[10] Ben Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction (Second Edition)*, Addison-Wesley, Reading, Massachusetts, 1992.

Delivery of High Quality Uncompressed Video over ATM to Windows NT Desktop

Sherali Zeadally
Department of Electrical Engineering
University of Southern California
University Park, DRB 116
Los Angeles, California 90089
Tel: 213-740-1450; Fax: 213-740-9280
zeadally@marco.usc.edu

Abstract

The emergence of high bandwidth applications such as medical visualization and virtual reality has exposed significant deficiencies in network, protocol, and end-system design. In this paper we discuss important end-system issues which arise when supporting applications demanding networked delivery and manipulation of *uncompressed* video to the desktop.

Our experimental network environment consists of DEC Alpha workstations using the Windows NT 4.0 operating system and connected via an ATM switch. We present the design and initial results of a network architecture that demonstrates the creation, manipulation, and distribution of high-quality uncompressed video using standard industry-based technologies. In addition, we discuss networking performance results and present a simple Windows Sockets 2.0 cost model for TCP/IP and UDP/IP over ATM.

An early potential market where this work is expected to have a direct impact is video editing in motion picture and television studios. In this context, we hope to provide cost-effective networked solutions aimed at replacing costly dedicated video editing hardware with the versatile capabilities of general purpose workstations and non-proprietary network solutions.

1. Introduction

Recent advances in network technologies and computer hardware have led to the development of powerful computers and high-speed networks. Along with these developments, a wide range of multimedia applications have also emerged, particularly those involving digital video and audio. Multimedia has become a critical technology for professional applications in hospitals, educational establishments, advertising agencies, and video studios. It is therefore crucial that emerging open-system solutions be able to support the creation, manipulation, distribution, storage, and retrieval of real-time multimedia data.

In the past, poor network bandwidth coupled with hardware limitations of workstation architectures prevented the support of uncompressed digital video to the desktop. However, the emergence of high-speed networks coupled with hardware improvements (e.g. CPUs, buses) is paving the way to supporting the high bandwidth requirements of those applications using digital video and high quality images. So far, approaches adopted by hardware/software designers have mainly been proprietary involving solutions such as special add-on cards or the use of special local buses to achieve high data transfer rates within the workstation. These approaches are expensive, inflexible, and very monolithic which makes it hard to modify or extend the underlying hardware/software design space. What is needed is an *open* architecture that exploits industry-based standards to

provide *cost-effective scalable* solutions for supporting high bandwidth multimedia applications such as those using uncompressed video.

The structure of this paper is organized as follows. Section 2 describes our motivation for the need for uncompressed video for certain applications. Section 3 presents our project goals. In section 4, we discuss the architectural design challenges that must be addressed to support full motion, high quality uncompressed video to the desktop and our proposed solutions. In addition, we also present networking performance results and a user-level architecture for flexible multimedia processing. Finally, section 5 makes some concluding remarks.

2. Motivation

In the past, the only way to capture, store, and deliver uncompressed video was to use dedicated proprietary systems such as digital disk recorders. These devices are not only limited in the amount of storage, but are highly specialized focusing on specific functions and therefore cannot be used as general purpose devices. Furthermore, such equipment is costly and difficult to upgrade. However, general purpose workstations do not suffer these serious limitations. They can be easily upgraded to take advantage of advances in new technologies such as faster networks, hardware improvements, and new operating systems.

To cope with hardware limitations such as disk access speed, memory access time, bus transfer capability, limited network bandwidth, and limited storage space, various compression schemes have been used for the storage and transmission of digital video over the network and its delivery inside the workstation. However, the majority of compression methods are based on lossy algorithms such as Motion-JPEG and MPEG [27][11]. The resulting video stream after a compression/decompression operation with lossy algorithms is of lower quality due to data loss during compression and added unwanted artifacts during decompression. The degradation in quality is tolerable and not apparent to the viewer for those applications where the compression/decompression cycle takes place only once. The problem arises when the video has to go through a series of intermediate compression/decompression cycles as is often the case with video editing/compositing in a post production process. In this case, the accumulated loss of data and artifacts become more obvious thereby causing considerable degradation in the original quality of the source video. This is illustrated in Figure 1 where a typical video editing session involves decompressing an incoming video stream into main memory, performing editing or adding special effects, compressing the modified video data before transmitting to the next workstation over the network. When several edit sessions are required, successive decompression/compression cycles are performed which degrade the quality of the original video stream.

Lossless algorithms commonly used by motion picture studios for editing typically produce around 2:1 compression for color images with moderately complex scenes. For instance, lossless JPEG works well only with continuous-tone images but does not provide useful compression of palette-color images [21]. Other lossless image compression algorithms such as ERIC and RICE also achieve around 1.4-1.96:1 compression ratios [14]. In reference [16], a mathematically lossless algorithm for Motion-JPEG is proposed which results in a compression factor of 1.6:1. Thus, it is obvious that lossless compression algorithms do not really give substantial decrease in data throughput (only by a factor of two). Therefore, we argue that rather than incurring the cost of minimal compression, a better and more effective solution is to use no compression at for all phases of video production: capture, creation, editing, assembly, and playback. In this way, we ensure the fidelity of the final product during the editing and composition stages of content creation and avoid quality degradation caused by compression/decompression cycles.

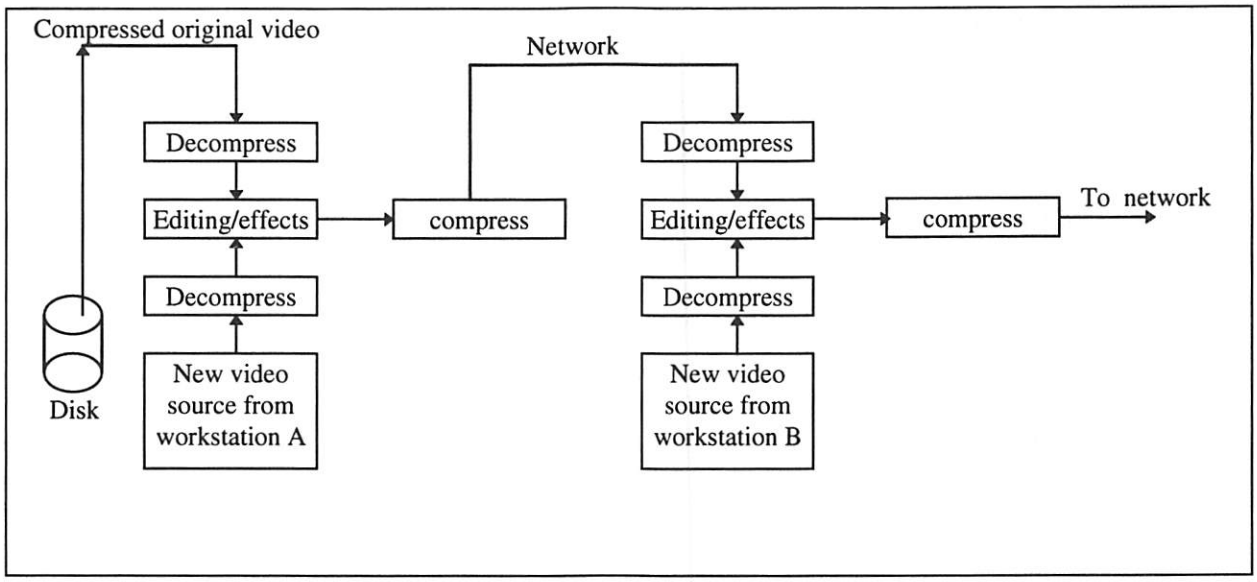


Figure 1 A typical workflow for editing/compositing/special effects requires multiple access to the original video data.

3. Project goals

One of the most important goals of the project is the design and implementation of a truly open architecture capable of delivering real-time, uncompressed, *high quality* digital video to the desktop. The use of the term high quality here means CCIR-601 studio quality video employed by most professional systems and movie studios. The digitized CCIR-601 [4] serial digital video signal contains 216 Megabits per second (Mbits/s) of video data (using 4:2:2 color encoding, 8 bits per sample at 27 MHz [28]). Ideally, we would like to perform all distribution and manipulation of high quality video in the digital domain. However, the lack of serial digital interface (e.g. serial digital interface cables for connecting a digital camera to frame grabbers) components means that we are forced to implement certain parts of the system using analog signals. For instance, in our prototype we are using an off-the-shelf camera for our live video source which delivers full-frame video to a frame grabber accepting an analog video input. Although we do not use CCIR-601 quality video directly, we are using a comparable data rate namely studio-quality digitized NTSC video which has a data transfer rate of 221 Mbits/s (30 frames, 640x480, 24 bit color pixel) compared to broadcast-quality NTSC which has a data rate of 120 Mbits/s. At present, the display is driven by analog RGB signals. In future, an all digital system might use Liquid Crystal Display (LCD) or digital light projector.

We plan to use Asynchronous Transfer Mode (ATM) (an international networking standard which transports network data as fixed-size 53-byte cells)[26] technology for the distribution and transmission of uncompressed digital video over the local area network. Another important goal is to provide a user-level architecture that allows flexible processing and manipulation on the video data stream. This is discussed in detail in section 4.4.

In order to achieve these goals, we are building an architecture based on industry standards including:

- General purpose workstations - DEC Alpha.
- Standard Microsoft Windows NT operating system.
- PCI bus architecture.
- Standard file format such as Audio/Video Interleave (AVI) and ActiveMovie [17].
- RAID hardware.
- ATM networking technology.

Operating system platform

One of the most crucial decisions we had to take early on in the course of this project was the choice of the operating system platform. We have chosen Windows NT due to a number of factors including: price, multiprocessor capability, its versatile networking capabilities such as the support for multiple protocols (TCP/IP, NFS, AppleTalk, DECnet, IPX/SPX, NetBEUI which allow connectivity to multiple platforms including UNIX-based systems, AppleTalk networks), true preemptive multitasking and multithreading, the availability of well-defined Application Programming Interfaces (APIs) including those supplied recently by Winsock 2.0 which allow applications to run natively and specify their Quality of Service (QoS) requirements on high-speed networks such as ATM, and access to numerous existing applications. Moreover, future porting of our software to other hardware platforms will be simpler since Windows NT already runs natively on other hardware platforms such as Intel.

4. Architectural design challenges

4.1 High performance video card

Content creation for broadcast-quality media involves a number of stages: pre-production (storyboards, script-writing, planning), production (animation, graphics, live video), and post-production (assembly and video editing). Each of the stages of content creation process requires distinct computer resources. We want to perform all of these steps in content creation using general-purpose high-performance workstations in a collaborative networked environment.

To perform all stages of content creation in the digital domain requires equipment such as cameras and video board (frame grabbers) to have the capability of delivering all media data digitally. Although digital cameras have now become available, there are no frame grabbers on the market that take an input digital video stream from such cameras and transfers the digitized video stream in *uncompressed* form. Consequently, we have chosen a frame grabber that accepts an analog video input such as that delivered by standard cameras. We have opted for a Coreco card [5] for our project since it is capable of delivering uncompressed digitized NTSC video stream at full-frame rate (640x480, 30 frames per second, 24 bit pixel) to the host. The Coreco board is a PCI-based adapter which has 2 MB on board video memory and an onboard VGA chip. We have not chosen other frame grabbers such as Matrox Meteor [15] (which can also deliver real-time full-motion video) because they need another display card such as MGA Millennium/Mystique [15] to work with since it has no on-board memory and also requires an additional PCI slot in the machine. At present the Coreco frame grabber works only on Intel-based architectures running Windows 3.1, Windows 95 or Windows NT 3.51/4.0, but will be ported to the DEC Alpha platform (with PCI slot) during the course of the project.

4.2 Bus bandwidth requirements

We are using DEC Alpha 600 Series high-performance uniprocessor workstations for our project. A simplified diagram of the DEC Alpha 600 Series architecture is illustrated in Figure 2. The Data switch implements the primary data path in the system and provides a 256-bit bus running at 33 MHz to main memory, 128-bit bus to the 21164 microprocessor and secondary cache, and a 64-bit bus running at 12 MHz to the CIA switch. The 21164 microprocessor has 3 on-chip caches: an 8 KiloByte (KB) primary cache (Dcache), an 8 KB primary instruction cache (Icache), and a 96 KB second level data and instruction cache (Scache). The CIA controls the Data switch, main memory, and interfaces to the 64-bit Peripheral Component Interconnect (PCI) local bus. In this Alpha model, there are three 64-bit and two 32-bit PCI slots [2].

The high data transfer rate associated with high quality video requires efficient transfers between the different components of the workstation. One of the crucial resources needed to deliver this high sustained data throughput is the bus bandwidth between the I/O devices and main memory. The DEC Alpha workstation architecture meets this requirement by a design which incorporates industry standard components such as the PCI bus which is capable of a maximum data transfer rate of over a gigabit per second for 32-bit PCI devices.

However, raw peak bus bandwidth alone is not enough for high sustained data throughput between network and user application. We plan to apply careful optimizations (e.g. efficient video/image display, the use of shared memory to reduce data copying) in the network-application data path at *all* levels namely network, operating system, and user level in order to achieve high end-to-end application throughput.

All peripheral cards (e.g. network, frame grabbers) used in this project are 32-bit PCI-based. Other features that led us to the choice of the PCI bus include support for multiple bus masters and the ability to perform device-to-device transfers (with no intermediate stops in memory) resulting in much more overlap between I/O and CPU operations. This can be exploited in teleconferencing applications where live video from a frame grabber can be routed directly to the network without the intervention of the CPU. However, the initial prototype architecture is not going to implement this feature. Instead, live uncompressed NTSC video will first be streamed to host memory and then from main memory to the network adapter.

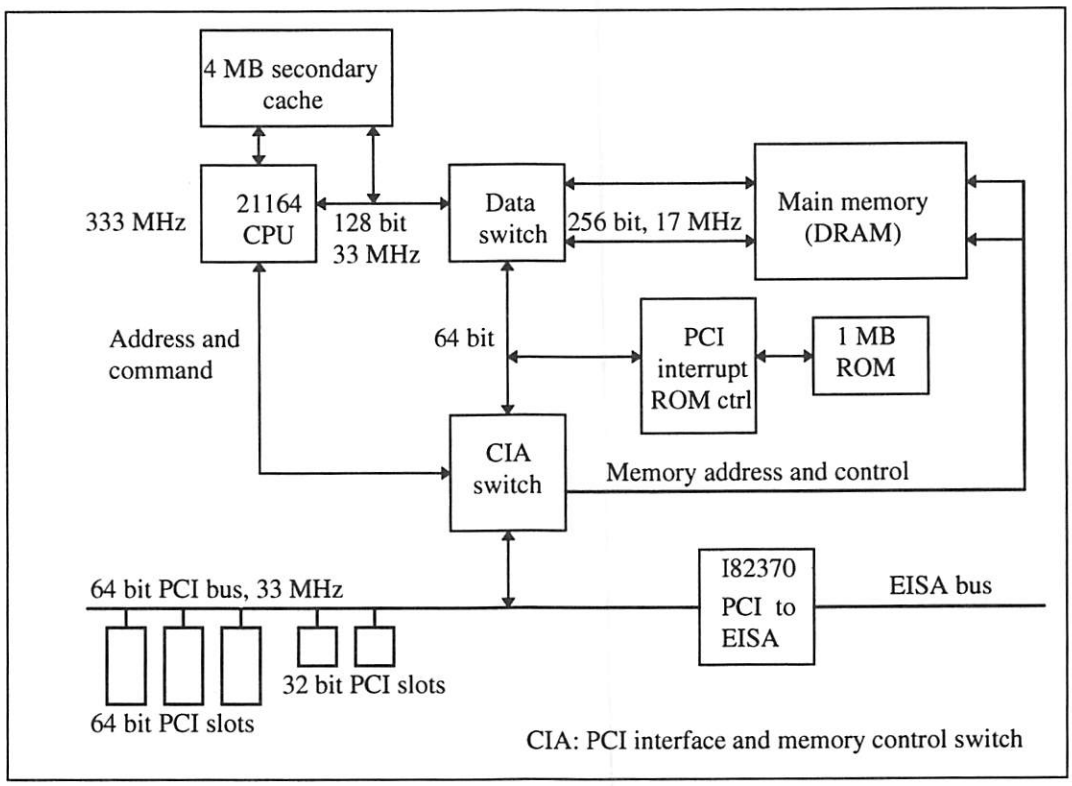


Figure 2 A simplified block diagram of the AlphaStation 600 Series [2].

4.3 Networking support

We have chosen ATM for our high-speed local area network testbed which consists of 4 DEC Alpha 600 workstations running Windows NT 4.0, each with a CPU speed of 333 MHz and 1 gigabyte of Random Access Memory (RAM). All workstations are connected to a DEC GIGAswitch/ATM [25] via multimode fiber. There are many reasons for the choice of ATM for our local area ATM network: although in this paper we focus only on one application area - editing involving uncompressed video, our long term goal however is to support different types of professional multimedia applications including interactive medical visualization, collaborative CAD, and interactive video collaboration. Each of these applications has its own traffic requirements (e.g. high throughput, low delay). ATM technology has the flexibility of providing the features necessary for different types of multimedia applications. For instance, ATM can provide QoS guarantees such as maximum cell rate, cell-transfer delay, or cell-loss ratio to user applications. Therefore, applications with different requirements will be able to negotiate their individual QoS requirements with the underlying network and be guaranteed a

certain level of performance. This in contrast to other high speed local area networks such as Fast Ethernet which operates on a "best effort" basis and provides no QoS guarantees. Windows Sockets 2.0 [29][30] includes support for direct access to ATM and allows new multimedia applications to take advantage of QoS features via an ATM Service Provider for an ATM network as illustrated in Figure 3.

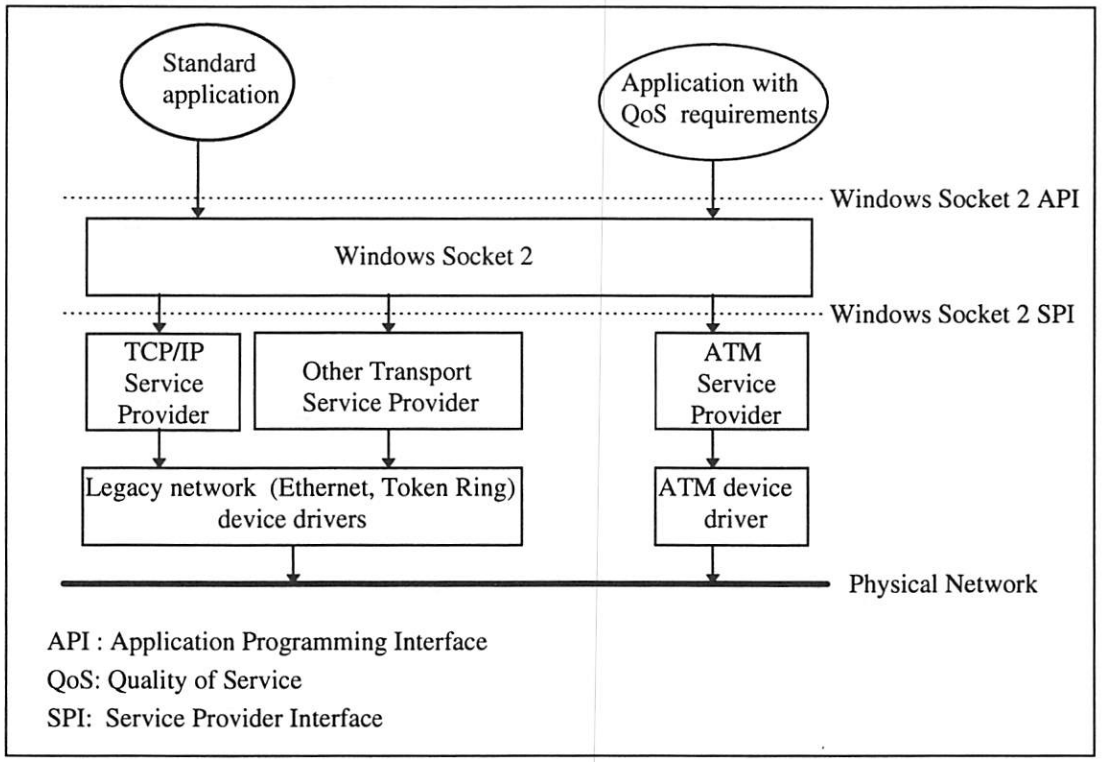


Figure 3 Windows Sockets 2 software support for multiple transport protocols and networks.

Windows Sockets 2 provides an API for user applications and a Service Provider Interface (SPI) for transport stacks. The Windows Sockets 2.0 API [29] includes mechanisms for applications to negotiate QoS with an underlying network such as ATM. Briefly, during connection establishment, a user application negotiates via Windows Sockets 2.0 the attributes of the connection. QoS attributes supported by Windows Sockets 2.0 API include: the source traffic description which describes (using parameters such as peak bandwidth) the way the application will send data over the network, upper limits on latency and latency variation acceptable, the level of service guarantee needed (the four levels defined are: absolute guarantee, controlled, predictive, or best effort), and cost for which is a metric is yet to be determined [29]. The QoS mechanism also allows applications to re-negotiate their QoS requirements after connection establishment using appropriate IOCTL calls.

Another important reason for the adoption of ATM is its ability to scale to higher speeds. The amount of bandwidth an application requires varies depending on the frame size, frame rate, and the quality of the image. Full-motion uncompressed studio quality NTSC video requires one-way sustained data transfer rate of 220 Mbits/s. We cannot satisfy this network bandwidth requirement with our current OC-3 ATM network which is only capable of transferring data at a signaling rate of 155.52 Mbits/s. Consequently, we have decided to choose a frame rate of 15 frames per second which requires a transfer rate of around 110 Mbits/s for our initial prototype architecture but as OC-12 (622.08 Mbits/s) becomes available, we intend to support full-frame rate of 30 frames per second. We do not believe that the cost of using OC-12 ATM will defeat one of our primary goals namely providing cost-effective solutions considering that motion picture studios incur significantly higher costs to effectively implement multiple editing sessions using expensive proprietary systems.

We have measured the throughput that can be delivered over ATM using conventional protocols such as TCP/IP and UDP/IP for a Classical IP [10] implementation - an ATM-aware layer below the traditional IP network layer which replaces the data link layer of the protocol stack (e.g. the media access control functions) with equivalent ATM functions.

For our experimental test configuration, we used two DEC Alpha workstations (each with a CPU speed of 333 MHz , 1 gigabyte of RAM, 4 gigabyte hard disk, and equipped with one ATM adapter (ATMworks 351 from DEC)) connected via the DEC switch using multimode fiber. All tests were conducted using Windows NT 4.0 and a socket buffer size of 64 KB for both TCP and UDP tests. We developed our own test programs for the throughput measurements. We measured the average application-application throughput between the two workstations connected via the DEC switch by timing bulk transfers from main memory over a sufficiently long period of time. All measurements were made using half-duplex connections. We also measured the CPU utilization using windows NT performance monitor [19] for the corresponding throughput achieved. The TCP/IP and UDP/IP results are shown in Figures 5 and 6 respectively. Note that the throughput result for TCP/IP is that obtained at the receiver - the transmitter throughput is the same (i.e. there were no dropped packets).

The results show that theoretical limit of 134 Mbits/s for OC-3 ATM is achieved on the DEC Alpha platform with Windows NT 4.0 for both TCP/IP (for message sizes above 40 KB) and UDP/IP. The dip in throughput for message size between 4 KB and 40 KB observed in Figure 5 is probably due to flow-control and acknowledgement algorithms used by TCP. There is a significant decrease in throughput at 8 KB. This is due to memory paging since 8 KB exceeds the amount of data that can be stored in a memory page. The page size used on the DEC Alpha 600 Series workstations is 8 KB but can hold less than 8 KB of data since it also keeps control information. Thus, with an 8 KB user message, the data has to be stored in two memory pages rather than only one memory page introducing extra memory overheads. CPU utilizations for TCP/IP and UDP/IP are around 55-65% and 50% respectively for reasonably large message sizes. In addition, we have also verified whether the CPU becomes the bottleneck at higher network speeds.

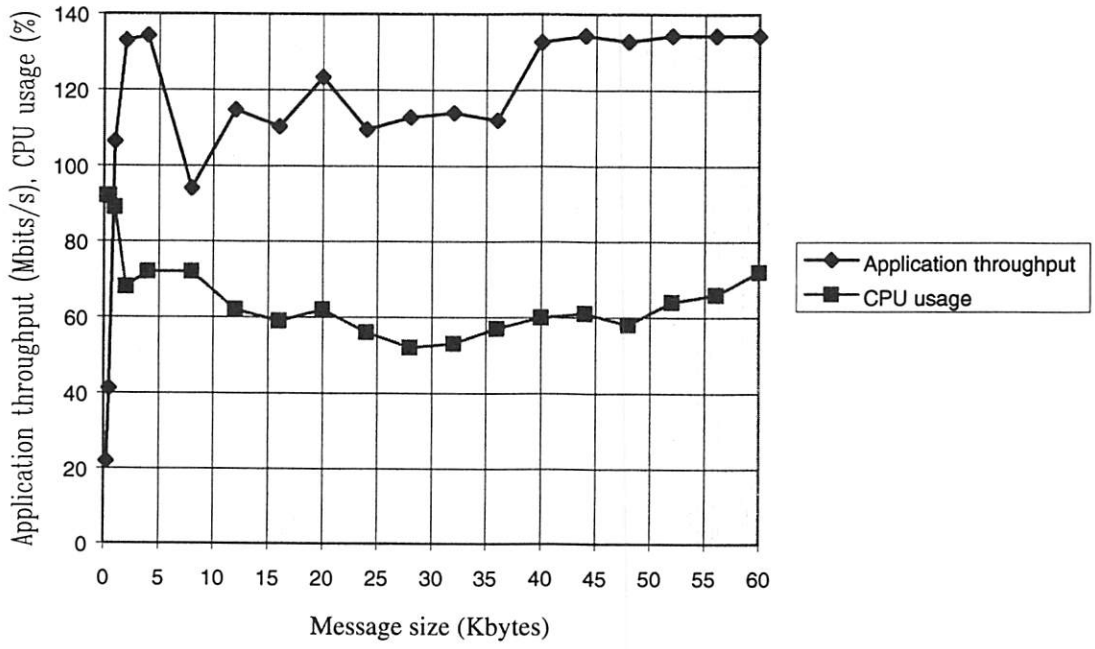


Figure 5 Variation of application throughput with message size for TCP/IP over ATM.

Using a full-duplex configuration (where each machine sends and receives data at the same time), we obtained an aggregate throughput of 240 Mbits/s for UDP/IP and the corresponding CPU utilization was around 90-95%. This clearly shows that at higher network speeds (e.g. OC-12 ATM), CPU becomes the bottleneck. The CPU availability factor becomes even more important since there are other operations that need to be performed in addition transferring data to user memory - the multimedia data needs to be displayed in addition to any intermediary processing that may be needed.

We are currently investigating how much CPU is required to display sustained full-frame (640x480, 24 bit color pixel) video. The software that allows grabbing of live video and transmission over the ATM network is still under development. This means that we have not yet been able to have a full data path from camera to the network and then to the display. However, the networking portion of the software has already been implemented, and allows a series of images stored in main memory to be transmitted over the ATM network and displayed at the receiving workstation. In a preliminary experiment, a series of high quality uncompressed video images (BITMAP - 640x480, 24 bit color pixel) stored in main memory at the sender were transmitted using TCP/IP over ATM, and the images were continuously displayed at the receiving workstation - a DEC Alpha 333 MHz. Thus, at the receiver data was being transmitted from the network adapter to main memory, and then from main memory to the graphics adapter which is a Digital PowerStorm 4D20 [9]. We obtained a CPU usage of almost 100% at the receiver. The frame rate observed after displaying the images was around 11 frames per second. This observed frame rate translates to a throughput of 81 Mbits/s. This is explained as follows: the throughput achieved when displaying image data read from the network is really made up of two components - the time it takes to read data from the network added to the time it takes to display the image. When either of these time components is high, this lowers the final throughput perceived. In our experiment, we achieved 120 Mbits/s throughput for data transfer between network adapter and main memory. Therefore, the decrease in overall throughput is due to the time it takes to display the image data. One of the factors that contributes to slowing down the display throughput is that the CPU is shared between displaying the images and handling incoming network data. That is, CPU cycles left over after handling network data are not sufficient to achieve higher display rates. Thus, CPU becomes the bottleneck. In this context, we would like to point out that the code for displaying the images is still in its early phase and has not really been optimized yet. As a result, we believe that there is still plenty of scope for improving the display performance and it is quite likely that the non-optimized display code has introduced unnecessary overheads causing high CPU consumption. We are currently focusing on understanding how the PowerStorm device driver works in order to optimize the data transfer path from main memory to the graphics buffer.

Based on these preliminary results, it is clear that there is a strong need for optimizing the display rate with as little CPU intervention as possible. One solution that might be worth exploring is the use multiple PCI buses - and possibly multiple processors as well - so that network data transfer between network adapter and main memory takes place in parallel with data traffic for the display on a separate PCI bus. A faster CPU will be needed not only for sustaining studio quality video data rate over the network but also for fast image/video displays. In this context, we are presently investigating methods that will optimize the display rate for high bandwidth applications.

The network throughput results show that Windows NT is **not** the factor responsible for the poor performance of TCP/IP and UDP/IP over ATM previously reported by other researchers [1][3]. The low performance (around 70 Mbits/s) achieved in [1][3] is due to several factors such as processor performance, network adapter hardware, ATM device drivers, and so on. Our results show that with careful integration of well implemented software (e.g. network device driver) and good hardware design (e.g. network adapter, host bus), it is possible to deliver high performance with Windows NT operating system. A more detailed discussion of networking performance of Windows NT 4.0 over ATM is given in [31].

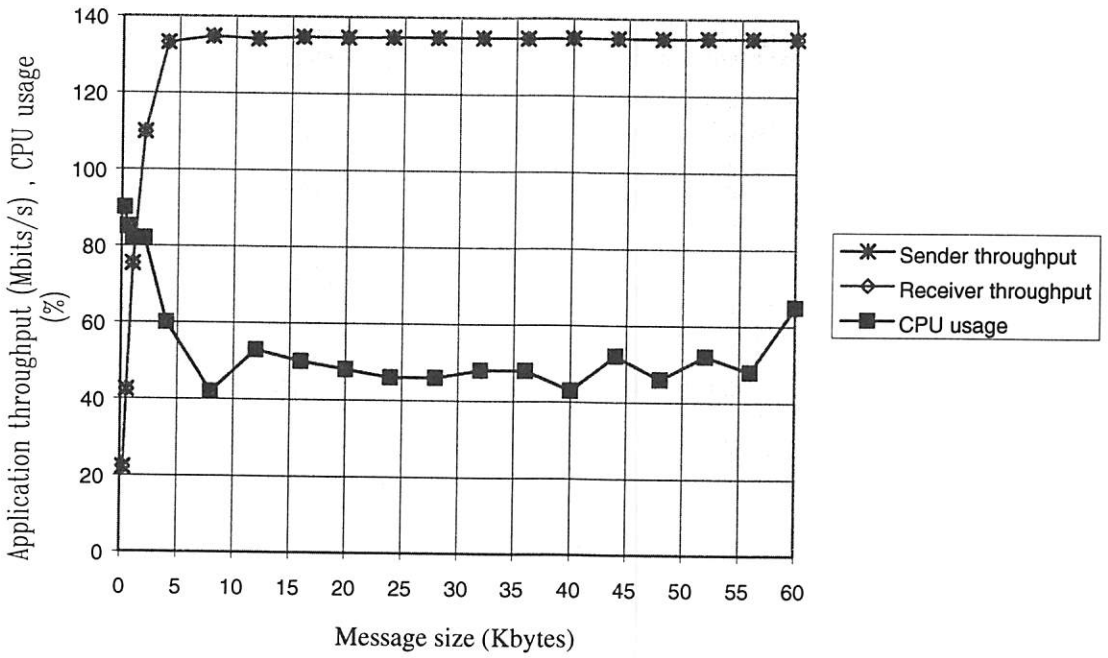


Figure 6 Variation of application throughput with message size for UDP/IP over ATM. The sender and receiver throughput graphs overlap.

4.3.1 A simple cost model for Windows Sockets 2.0

Based on the results obtained from Figures 5 and 6, a simple cost model can be derived for Windows Sockets 2.0 as follows:

The 21164 microprocessor on the DEC Alpha 600 Series can issue two integer/memory class instructions and two floating point instructions for every clock cycle [2][24]. However, experimental measurements performed with several applications in [7] have demonstrated that two clock cycles per instruction for the processor is more likely. Therefore, using the result in [7] for the number of clock cycles per instruction, a 333 MHz clock (used in our experiments) implies a processor performance of 167 MIPS.

In order to obtain the fixed cost that has to be incurred by the underlying socket implementation protocol, and other operating system overheads, we measured the latency for a 4 byte message which was transmitted over the ATM network. The actual measurement made was the round-trip delay and the latency was then calculated by halving the round-trip result. We obtained 178 microseconds for TCP and 84 microseconds for UDP. In these measurements, we ignored the fiber delay since the workstations used were separated by a small distance (only a few meters).

Cost for TCP/IP message over ATM:

Latency for TCP: 178 microseconds.
Fixed message cost = 178 * 167 = 29726 instructions.

From Figure 5, we observed an average CPU usage of 60% corresponding to a throughput of 134 Mbits/s (i.e 16.75 megabytes/second).

Number of instructions used to achieve 16.75 megabytes/second = 167 * 0.6 = 100.2 MIPS.
Number of instructions per byte = 100.2/16.75 = 5.98 ~ 6 .

Therefore, cost of message for TCP = 29726 + 6 instructions per byte.

Cost for UDP/IP message over ATM:

Latency for UDP: 84 microseconds.
Fixed message cost = 84 * 167 = 14028 instructions.

From Figure 6, we observed an average CPU usage of 50% corresponding to a throughput of 134 Mbits/s (i.e. 16.75 megabytes/second).

Number of instructions used to achieve 16.75 megabytes/second = 167 * 0.5 = 83.5 MIPS.
Number of instructions per byte = 83.5/16.75 = 4.98 ~ 5 .

Hence, cost of message for UDP = 14028 + 5 instructions per byte.

The above results show that UDP gives 17% better performance over ATM compared to TCP on a per byte basis. The fixed cost is twice for TCP compared to UDP. It has not been possible to get similar results for native ATM (i.e. without using protocols such as TCP/IP or UDP/IP) since the ATM device driver used does not support it for user applications. However, this will be done in future work.

4.4 Manipulation/processing of multimedia data

The first generation of multimedia applications have been mostly based on simply *presenting* multimedia data to the end-user with little or no processing (e.g. video display for a video conferencing application). However, the second generation of multimedia applications will not only involve mere presentation of multimedia data but also *manipulation/processing* on either all the media data stream or specific portions of it. Our goal is to have a flexible and extensible architecture capable of processing different media types in a modular fashion. In this context we are developing a User-level Multimedia Module (UMM) which allows full user control on the data path of a video stream originating either from the network (from another workstation) or from a live source such as a camera as shown in Figure 7.

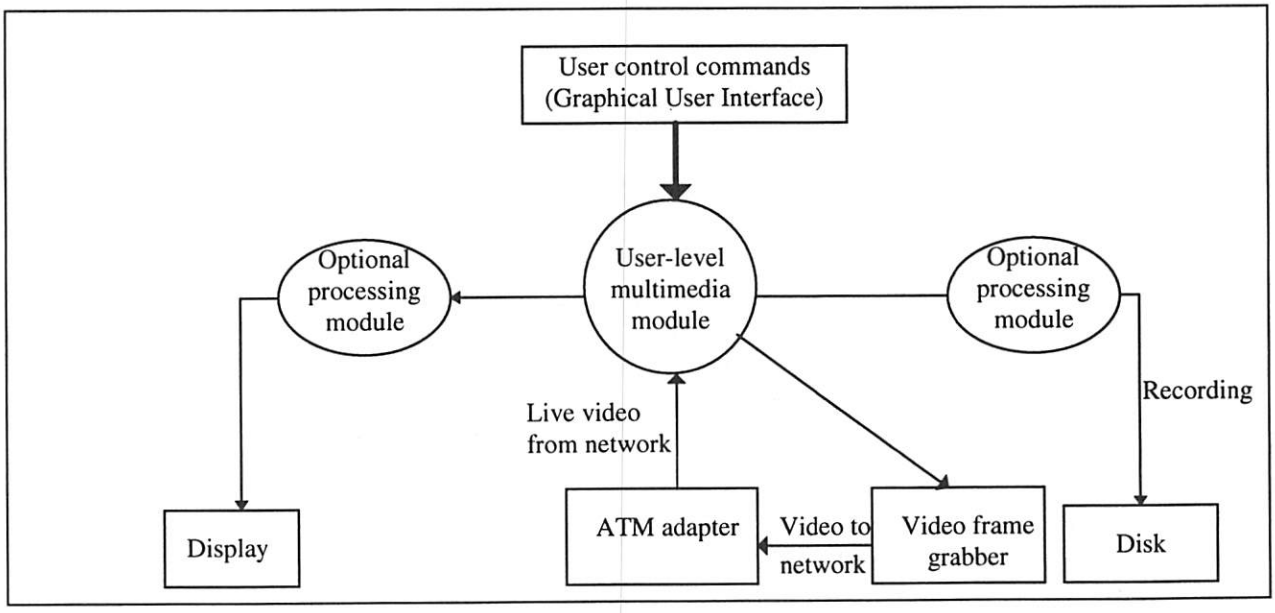


Figure 7 A functional description of the User-level Multimedia Module (UMM) and its interaction with display, network, and storage devices.

Two common scenarios are depicted for a receiver for video coming from the network: the user may choose to display/save all or part the video data (with or without intermediary media data manipulation using optional processing modules). An example of an application where such functionality is necessary is one which performs tracking of features or objects in a video stream, and takes appropriate actions (such as saving to a file or displaying only a specific object) based on the actual media content. Figure 7 also shows the case for a transmitter whereby live video can be grabbed using the frame grabber and routed to the network adapter for transmission. We have already implemented this portion of the UMM. To simplify integration of the UMM with other components of the system, we have chosen ActiveMovie as our design space. Some of the major features that make ActiveMovie an attractive choice include: the support of multiple multimedia data types (e.g. digital video from various types of codecs, still images, digital audio, and so on), high data throughput can be achieved using large buffers (compared to Video for Windows which limits transfers to a maximum of 64 KB), major operating system overheads such as CPU data copying between applications are eliminated using shared memory, and low-level synchronization support (i.e. at field level rather than frame level) which ensures timely delivery of data to presentation devices. Another interesting feature of ActiveMovie is the use of processing elements known as *filters* which can be connected to each other (thereby giving what is called a *filter graph*) to provide a required processing sequence. Thus, in Figure 7, the processing elements will be implemented as filters, and the various functions of the UMM itself will really be implemented as a filter graph.

So far, we have successfully demonstrated the capability of the Coreco cards for grabbing full-frame live video in real-time host memory with the option of saving to disk. The capability of sustaining full-frame uncompressed video to main memory has shown that the PCI bus is capable of providing the bandwidth required. We have enhanced the Coreco software to include the capability to save live uncompressed video clips as AVI files. However, since high throughput is limited by the AVI format, we are presently developing new software that will exploit the use of OpenDML [12] format which should allow higher playback rates than AVI format.

4.5 Video transmission format

The actual format to be used for transporting uncompressed video over our ATM network has not yet been decided. The advent of ActiveMovie Streaming Format (ASF) [18] as an open and extensible data-independent format for storing and transmitting multimedia content over a wide range of networks appears promising and is one of the options we are presently considering. However, the lack of information on the specification of ASF has made it hard to come to a decision on this issue. Furthermore, it is also not clear whether ASF will be able to provide the high sustained data rate required for uncompressed video. Meanwhile, we are currently looking at other possible transport format for video.

4.6 Storage and video playback

Uncompressed video requires a large amount of storage. For instance, a 45 seconds uncompressed video stream uses one gigabyte of storage space. Uncompressed video playback from storage disks also needs high sustained throughput - typically 220 Mbits/s for full-frame, full-rate digital video stream. Initially, we are using five 4 gigabyte Fast Wide SCSI II disk drives for storage space. The current focus on the delivery and manipulation of live video rather than investigating issues such as video playback issues from video storage servers. This has been dealt by other researchers [13][20][23] and is not the subject of our research. Yet, if we store uncompressed video, we will need to achieve a reasonable data rate for video playback. Initial experiments on a Fast Wide SCSI II disk gave a throughput of about 50 Mbits/s. This is clearly insufficient to provide the necessary data rate (220 Mbits/s) required for uncompressed video. One option is to use faster SCSII disk controllers such as UltraSCSI which results in a peak data rate of 40 Megabytes per second and multiple disk drives configured to use software stripe sets supported by the NT File System (NTFS [6]). In order to achieve the required data rate for an uncompressed video stream, striping has to be performed across controllers. Another option that will be investigated is the use of a hardware RAID [22] system such as Digital StorageWorks [8] to provide the required throughput.

5. Conclusions

In this paper, we have described our motivations behind the need for the delivery of high-end *uncompressed* video to the desktop. We have outlined the major architectural design issues that need to be addressed to achieve this goal. We have presented solutions we are currently implementing to solve the architectural challenges. Our design approach is based on an open architecture which exploits existing industry-based standard technologies rather than the use of proprietary hardware or software. This will allow cost-effective, scalable, networked solutions for multimedia applications that use high-end digital video (e.g. video editing in motion picture studios). In addition, we have also reported application-application throughput reaching theoretical limit for TCP-UDP/IP over ATM and presented a simple cost model for Windows Sockets 2. The results demonstrate that Windows NT is a suitable operating system choice for meeting the high performance requirements of applications running over high-speed networks such as ATM.

Acknowledgements

The work presented in this paper is the result of a joint effort between Digital Equipment Corporation (Massachusetts) and a group of investigators in the NSF-sponsored Integrated Media Systems Center at the University of Southern California which is investigating the manipulation and distribution of full-motion, high quality *uncompressed* video to the desktop (running Windows NT 4.0) over an ATM network.

The author expresses his gratitude to Jim Gray who provided valuable feedback and ideas on early drafts of this paper and whose patience and support throughout the revision process are greatly appreciated. The author gratefully acknowledges suggestions from Tony Levi on an earlier version of this paper. The author thanks Grig Gheorghiu for his comments on the paper and the many employees of Digital Equipment Corporation (Massachussetts) for their support, in particular Doug Washabaugh for his valuable discussions on many aspects of this work. The author also expresses his gratitude to the anonymous reviewers for their valuable ideas. The USC work is supported by the Integrated Media Systems Center NSF Grant EEC-9529-152.

References

[1] Andrikopoulos I. et al. "TCP/IP Throughput Performance Evaluation for ATM Local Area Networks." In Proc. of 4th IFIP Workshop on Performance Modelling and Evaluation of ATM Networks, Ilkley, July 1996.

[2] Bhandarkar D. "Alpha Implementations and Architecture, Complete Reference and Guide." Digital Press, ISBN 1-55558-130-7, 1996.

[3] Carbone J. "Preliminary Network Performance Study of a Windows NT ATM Network." Internal Report, Odyssea Systems Corporation, June 1996.

[4] CCIR Recommendation 601-2. "Encoding Parameters of Digital Televison for Studios." Vol. XI - Part 1, International Telecommunications Union, Geneva, 1990.

[5] Coreco Inc. "The Oculus Driver Command Interpreter Manual." Edition 1.0, Revision 0, Coreco Inc, St. Laurent, Quebec, Canada, 1990.

[6] Custer H. "Inside Windows NT." Microsoft Press, ISBN 1-55615-481-X, 1992.

[7] Cvetanovic Z. and Donaldson D. "Alpha Server 4100 Performance Characterization." Digital Technical Journal, Vol. 8, No. 4, 1997. Http://www.europe.digital.com/info/DTJ001/DTJ001PF.PDF.

[8] Digital Equipment Corporation. "StorageWorks Solutions 7 device, 16-bit Deskside, Expansion Pedestals, BA356-K Series User's Guide." Digital Equipment Corporation, Massachusetts, December 1995.

[9] Digital Equipment Corporation. Http://www.alphastation.digital.com/products/powerstorm/pssld13.html.

[10] Laubach M. "Classical IP and ARP over ATM." RFC 1577, Network Working Group Internet Draft, January 1994.

[11] LeGall D. "MPEG: A Video Compression Standard for Multimedia Applications." CACM, Vol. 34, No. 4, April 1991.

[12]Legault A. and Matey J. "OpenDML AVI File Format Extensions, Version 1.02." http://www.matrox.com/videoweb/odml_826.html, February 1996.

[13] Lougher P. and Shepherd D. "The design of a storage server for continuous media." Computing Journal, Vol. 36, page 32-42.

[14] Majani E. "Lossless compression." http://www-ias.jpl.nasa.gov/HPCC/eric/node4.html.

[15] Matrox Inc. "Display Capabilities with Matrox Meteor and MGA Millennium/Mystique." http://www.matrox.com/imgweb/met_mga.html.

[16] Matrox Inc. "Mathematically Lossless Motion JPEG - Better Than Uncompressed Video." http://www.matrox.com/videoweb/hot_math.html.

[17] Microsoft Inc. "Microsoft ActiveMovie 1.0 Software Development Kit." October 1996.

[18] Microsoft Inc. http://www.microsoft.com/corpinfo/press/1996/mar96/pronetpr.html, 1996.

[19] Microsoft Inc. "Microsoft Windows NT Resource Kit." Microsoft Press, ISBN 1-57231-343-9.

[20] Mourad A. "Issues in the design of a storage server for video-on-demand." Multimedia Systems, pages 70-86, Vol. 4, 1996.

[21] Oxford University. "JPEG image compression." Oxford University Libraries Automation Service, http://www.lib.ox.ac.uk/internet/news/faq/archive/jpeg-faq.part1.html.

[22] Patterson D., Gibson G., and Katz R. "A case for redundant arrays of inexpensive disks (RAID)." In Proc. of the ACM SIGMOD 1988, Chicago, Illinois, pages 109-116.

[23] Reddy A. L. N. and Wyllie J. "Disk scheduling in a multimedia I/O system." In Proc. of the ACM Multimedia , pages 225-233, Anaheim, CA, 1993.

[24] Sano B. Private Communications.

[25] Souza R. et al. "The GIGAswitch System: A High-Performance Packet Switching Platform." Digital Technical Journal, Vol. 6, No. 1, 1994.

[26] Vetter R. "ATM concepts, architectures, and protocols." CACM, Vol. 38, No. 2, pages 30-38, 1995.

[27] Wallace, G. "The JPEG still-picture compression standard." CACM, Vol. 34, No. 4, April 1991.

[28] Watkinson J. "The Art of Digital Video." Butterworth-Heinemann Ltd. ISBN 0-240-51369-X, pp. 399-411, 1994.

[29] Stardust Technologies Inc. "Windows Sockets 2, Application Programming Interface, Revision 2.2.0." http://www.stardust.com/wsresource/winsock2/ws2sdk.html/wsapi22.doc, May 1996.

[30] Stardust Technologies Inc. "Windows Sockets 2, Service Provider Interface, Revision 2.2.0." http://www.stardust.com/ wsresource/winsock2/ws2sdk.html/wsspi22.doc, May 1996.

[31] Zeadally S. "TCP-UDP/IP Performance over ATM on Windows NT." In Proc. of 3rd IEEE ATM'97 Workshop, Lisbon, Portugal, May 1997.

Dreams in a Nutshell

Steven Sommer
Microsoft Research Institute and Department of Computing,
School of Mathematics, Physics, Computing and Electronics,
Macquarie University,
NSW 2109, Australia

steve@mpce.mq.edu.au

Abstract

The Dreams extensions have been developed in order to support distributed real-time applications within the conventional operating system paradigm. To demonstrate the viability of the extensions, they have been implemented within Windows NT. This paper introduces the important components of the Dreams extensions, provides an overview of the implementation, and highlights some of the experiences gained from the implementation.

1. Introduction

The aim of the Dreams (Distributed Real-Time Extensions with Application to Multimedia Systems) project is to provide a complete set of extensions for conventional operating systems, so that they may support real-time and distributed real-time processes within the conventional operating system paradigm. To demonstrate the viability of our extensions we have partially implemented our extensions within the Windows NT™ [2] operating system. This paper introduces and motivates the inclusion of each of the key components of the extensions, provides a brief overview of their implementation, and highlights some of the experiences gained from the implementation.

A fundamental component of the conventional operating system paradigm is the ability to run independent applications simultaneously while protecting these applications from interfering with one another. This is quite different from the paradigm of real-time systems, where all tasks work together with a common aim. The Dreams extensions allow real-time applications to be protected from one another.

Unlike traditional real-time systems [1, 4, 6], conventional operating systems do not have *a priori* knowledge of the arrival times and behaviors of real-time tasks; the schedulability test and scheduling algorithm must be performed on-line. Conventional operating system also allow: interrupts which run at a higher priority than both non-real-time and real-time applications; subsystems which execute system calls but which are otherwise indistinguishable from user applications; and dynamic distribution of applications.

To maximize the ease of adoption of these extensions, they have been developed to have a minimal impact on the code and conventional behavior of the operating system and on the programming model used to develop real-time applications. The Dreams extensions have been designed to be independent of any particular operating system.

In [8], we identified a new real-time process abstraction, called the *transient periodic process,* which is better suited to real-time applications running on a conventional operating system. Transient periodic processes have two distinguishing features.

- They act as periodic processes while running but, unlike traditional periodic processes, an entire process may start and complete at any time.
- The starting time of the first invocation of the transient periodic process is not constrained by the process.

An overview of the Dreams model, the advantages of the transient periodic process abstraction, and a general comparison with other similar models, can also be found in [8].

2. Protection

There are two major areas in which a conventional operating system must be enhanced to allow it to protect real-time applications from one another.

Firstly, the operating system must offer a new form of protection, called *temporal protection*. This encompasses the requirement that the timing behavior of one task should not be able to affect the ability of another, independent task to meet its deadline. To achieve this, the scheduling method needs to be altered, the timing behavior of each task must be monitored and enforced, and a method for effectively dealing with overrun tasks must be developed. Details of the Dreams approach to temporal protection can be found in [9].

Secondly, the operating system's support for resource sharing must be extended to be consistent with the requirements of real-time scheduling. The areas of resource sharing that must be considered are the concurrency support primitives (for example, a mutex) and the subsystem call mechanism. There are two predominant alterations required in this area. The first is to ensure that if the real-time task that should be running is blocked on another task, then the other task is immediately executed. In [7] we detail our approach for achieving this, that is, an extended form of priority inheritance [5], and argue for its inclusion within all conventional operating systems. We have also developed an extension to priority inheritance, called quantum inheritance, which also improves the conventional functioning of the operating system. The second resource sharing alteration consists of ensuring that a real-time task's blocking time is bounded and that the blocking can be effectively modeled within the schedulability test.

3. Modeling the Operating System

A fundamental element of the Dreams model is the guarantee that an application that does not exceed its reserved time will always meet its deadlines. The Dreams model ensures that the system as a whole has sufficient capacity to satisfy all of the active real-time applications by using an admission mechanism and by constraining and modeling particular parts of the operating system.

Many aspects of the system are outside of the control of the Dreams scheduler. The schedulability test must include the interference of the system as well as the system's own resource requirements. Elements of the system which impact real-time threads include hardware and software interrupts, caching, deferred procedure calls, parts of the conventional system, and the Dreams scheduler itself. Our schedulability test also models the impact of interrupts on effective enforcement and preemption.

We have formally extended Liu and Layland's earliest deadline first (EDF) schedulability test [3] to include clocked, interspaced sporadic, and bursty sporadic real-time interrupts. We have further extended the tests to include priority inheritance and critical sections, both with and without enforcement mechanisms. Our schedulability results can be found in [10].

Some modifications to the conventional operating system are necessary to bound the system's behavior; for example, interrupts and deferred procedure calls must have a maximum duration and frequency, or they must tolerate occasional suspension. Particular system calls must have a bounded timing behavior so that they can be used by real-time tasks. Our work in this area is not yet complete.

4. Distribution

The Dreams model allows a transient periodic process to be distributed to networked machines when there are insufficient resources to run the process on the local machine. The distribution component is responsible for the selection of the networked machines and the distribution of the transient periodic process to those machines. The distribution component of Dreams has not yet been integrated into our implementation.

Most of the difficulties involved in supporting distributed real-time processes are also key concerns in the research areas of real-time communication, process distribution, and load balancing. Although we have briefly examined the important issues required for supporting distributed real-time processes, we have chosen to concentrate primarily on those issues which should be addressed differently in the Dreams context.

We have developed a distributed placement algorithm for selecting the remote machines on which the transient periodic processes should execute. The

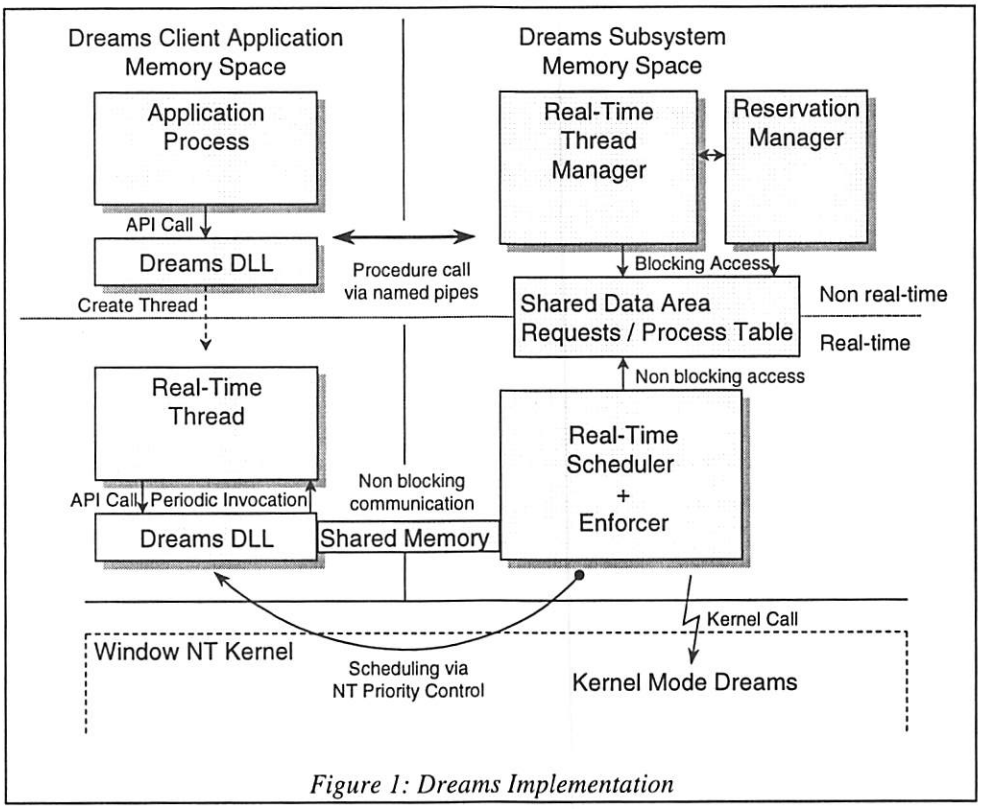


Figure 1: Dreams Implementation

algorithm was developed for optimal performance within the Dreams framework. Currently, we have only analyzed the algorithm within a simulation environment.

The inclusion of distribution within the Dreams model has had a significant influence over the design of many of the other components of the model. For example, one of the most compelling reasons for strictly preventing system overload comes from distribution. Distribution requires that network card interrupts be serviced in a real-time manner; sporadic interrupts must be enabled and modeled within the schedulability test. The system must be able to distribute each real-time entity separately; the real-time entity must, therefore, be implemented as a thread or process, not as a block of code. One of the requirements for the transient periodic process abstraction was that it allowed distribution to be performed in a manner consistent with the functionality of an operating system containing the real-time Dreams extensions.

5. Implementation Design

We have partially implemented the Dreams extensions within version 3.51 of the Windows NT operating system. A real-time[†] application is implemented as a Win32™ application with additional real-time facilities. Most of the new facilities are provided by a user-mode process called the *Dreams subsystem*. To use these facilities the application need only include an additional library and header file.

The transient periodic process of our model has been implemented within Windows NT as a Win32 thread. An application creates a real-time thread in the same manner as creating a conventional thread but with additional parameters specifying the real-time properties of the thread. These include its start time, reserved time, deadline, period, and additional flags.

Figure 1 provides an overview of the implementation. The following paragraphs highlight the important

[†] Our use of the term *real-time* is unrelated to the "real-time" priority class of Windows NT.

components in the figure, by tracing though the creation and initial execution of a real-time thread.

When an application makes an API (Application Programming Interface) call to create a real-time thread, the Dreams DLL (Dynamic Link Library) sends the request to the real-time thread manager, which runs as part of the Dreams subsystem. If the new task set passes the schedulability test performed by the reservation manager, the real-time thread manager approves the request. If not, the real-time thread manager may attempt to distribute the task by communicating with other thread managers. If the task can be run locally, the DLL creates a new thread and an associated shared memory segment. The new thread performs its initialization and then waits for its first invocation. The DLL then passes handles and control of the thread to the subsystem. The real-time thread manager performs the setup required for the thread in the subsystem and places a token for the thread in the real-time scheduler's shared data area. The scheduler takes control of the new real-time thread when it has spare time.

The scheduler executes at the highest Windows NT priority. It allocates CPU time to a real-time thread by making it the second highest priority thread. The scheduler and real-time thread communicate using the shared memory area. The periodicity of the real-time thread is implemented within the Dreams DLL by having a single Win32 thread call the specified entrance point of the real-time thread at the beginning of each invocation. Each invocation completes by making an API call which signals the completion to the scheduler.

6. Windows NT Implementation Experiences

In this section, we highlight some of the experiences gained from the implementation of the Dreams extensions within Windows NT.

An important issue in the design of the implementation was the choice of placing the extensions in a subsystem, or placing them in the Windows NT executive and kernel. We decided to place the extensions in a subsystem and to move components into the kernel as it became necessary. Having the extensions in a subsystem was consistent with our goals of minimizing the effect on the conventional functioning of the operating system and minimizing the alterations to the code of the existing operating system. Adopting this approach allowed the extensions to be developed, modified, tested, debugged, displayed, and understood, far more easily than if they were placed in the kernel.

The Dreams scheduler was the one component that could suffer from being placed in a subsystem. Placing the Dreams scheduler in the subsystem introduces a performance overhead due to additional context switches, and has the potential for duplicating kernel scheduling information. The overhead of invoking the Dreams scheduler is equivalent to a single subsystem call. This additional overhead is of no consequence to the other components of the model. In our current implementation, no data is shared between the two schedulers.

The one problem that we found in implementing the functionality of the scheduler in the subsystem was that the Dreams scheduler was occasionally being invoked late. This was seen in two forms. Firstly, the scheduler could be invoked a full millisecond late. It could detect this at the time it was invoked. Secondly, the scheduler could be invoked near the end of a one millisecond interval instead of at a one millisecond boundary. It could detect this by noting that the time had changed during the interval within which it had been executing. The scheduler was invoked late for a number of reasons. The scheduler used a Win32 multimedia timer to control the time that it was next invoked. Often the timer and the scheduler were delayed by the execution of system functions; these should not have taken a full millisecond. Unfortunately, the time at which the timer should fire is specified as a number of milliseconds from the current time. The current time could change while the call to set the timer was being made, causing the timer to fire one millisecond late. Finally, the time set by the clock interrupt, the time used by the Win32 timers, and the time obtained from interrogating the hardware clock were all slightly out of phase; this could also cause the Win32 timer to fire one millisecond late. To rectify this problem, we decided to invoke the Dreams scheduler off the clock interrupt in a similar manner to that which occurs at a quantum end.

Placing the Dreams scheduler in the subsystem led us to develop a two tiered scheduling approach that, with the implementation of priority inheritance within the priority scheduler, turned out to be remarkably advantageous. The priority scheduler, with priority inheritance, ensures that the correct thread is

scheduled each time the real-time thread blocks or makes a subsystem call. The Dreams scheduler need only be concerned with implementing the real-time scheduling algorithm. When it decides to schedule a different real-time thread, it simply drops the priority of the real-time thread being preempted and raises the priority of the next thread to be scheduled. The priority inheritance protocol will then transparently and precisely implement the desired scheduling behavior, *even if a different thread was actually executing*: for example, if the executing thread was a non-real-time thread completing a critical section, so as to unblock a subsystem thread which was performing a system call on behalf of the scheduled real-time thread. If, in this example, the real-time scheduler later decides to schedule the preempted real-time thread, then the non-real-time thread that had actually been preempted will automatically be executed to release its critical section and allow the system call to complete. The complexity of this scheduling can be hidden from the Dreams scheduler, the real-time task computational usage tracking, the enforcement mechanism, and the schedulability model.

We chose to implement priority inheritance within the QLPC (Quick Local Procedure Call) mechanism as this mechanism appeared to be the most frequent cause of priority inversion [5]. The implementation consisted of: removing the existing scheme for partially overcoming some of the effects of priority inversion; adding a smaller amount of code to implement priority (and quantum) inheritance; and lowering the priority of the Win32 subsystem servicing threads to the idle priority. Significantly more effort would have been required to implement priority inheritance in a single tiered scheduler in which one of the priority queues was being used to schedule threads in an EDF manner.

The benefits from the implementation of priority inheritance were not limited to the Dreams scheduler. After implementing priority inheritance, we found that a number of very poor scheduling behaviors were removed from the system. As a result of the implementation, an optimization used within Windows NT for particular combinations of priorities was usable for threads of all priorities. This yielded a performance increase for GDI calls belonging to threads at particular priorities. We ran two video display applications at the same priority. On the system containing our modifications, the videos played more smoothly; they appeared to execute at the same time, rather than one after the other. Finally, we found that a lower priority video player no longer interfered with the execution of a higher priority editor.

7. Conclusion

This paper has introduced the key components of the Dreams extensions. The extensions are required to allow a conventional operating system to support competing real-time applications within the conventional operating system paradigm. Most of the extensions described in this paper have been successfully applied in our Windows NT implementation. This paper has also provided an overview of the implementation and has highlighted some of the experiences gained from the implementation.

Acknowledgments

I would like to thank John Potter, Mark Dras, and Yan Han for their assistance with this paper. This work was supported by a Microsoft Research Institute Fellowship and an Australian Postgraduate Research Award.

References

[1] Burns, A. and Wellings, A. *Real-Time Systems and their Programming Languages*, Second Edition, Addison-Wesley, 1996.

[2] Custer, H. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[3] Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM* 20(1):44-61, January 1993.

[4] Ramamritham, K. and Stankovic, J. Scheduling Algorithms and Operating Systems Support for Real-Time Systems, *Proceedings of the IEEE* 82(1):55-67, January 1994.

[5] Sha, L., Rajkumar, R. and Lehoczky, J. Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers* 36 (9):1175-1185, 1990.

[6] Shin, K. and Ramanatham P. Real-Time Computing: A New Discipline of Computer Science and Engineering, *Proceedings of the IEEE* 82(1):6-24, 1994.

[7] Sommer, S. Removing Priority Inversion from an Operating System. In *Proceedings of the Nineteenth Australasian Computer Science Conference*, 131-139, January 1996.

[8] Sommer, S. and Potter, J. Operating System Extensions for Dynamic Real-Time Applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[9] Sommer, S. Temporal Protection in Dreams. In *Proceedings of the Twentieth Australasian Computer Science Conference*, 56-64, February 1997.

[10] Sommer, S. and Potter, J. *Admissibility Tests for Interrupted Earliest Deadline First Scheduling with Priority Inheritance,* Technical Report C/TR97-10, MRI, MPCE, Macquarie University, 1997.

Adding Response Time Measurement of CIFS File Server Performance to NetBench

Karl L. Swartz - kls@netapp.com
Network Appliance

Abstract

The standard benchmark for NFS file server performance, SPEC SFS (also known as LADDIS), measures performance in terms of both throughput—the aggregate amount of data a file server can move across the network per unit of time—and response time—the time required to service an individual client request. NetBench, the most commonly used file server benchmark for the CIFS (or SMB) protocol measures only throughput. Network Appliance believes response time is as important a performance metric as throughput, especially in the highly interactive environment typical of CIFS networks, since throughput offers little solace to a user waiting to access a file.

This paper documents the methodology and tools developed to measure response time during a NetBench run. While cumbersome and primitive, useful data has been produced, demonstrating that the fundamental idea is sound. SPEC SFS has had a noticeable effect on vendors of NFS file servers, motivating them to improve response time from an average of 50ms in 1993 to less than 10ms in 1997. Given the ability to measure response time in the CIFS environment, hopefully a similar improvement can be encouraged in CIFS file servers.

1. Introduction

> *"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."*
> - Andrew Tanenbaum

Throughput is an important metric for file server performance, but for individual, interactive users, good response time is far more critical. The standard benchmark for NFS file servers, SPEC SFS (also known as LADDIS) [1,2], measures both. When Network Appliance introduced support for the CIFS (or SMB[1]) protocol [3], we wanted to measure both attributes of our file server's performance using this new protocol, but were disappointed to find that the file server portion of NetBench [4][2] only measures throughput.

The desire to have response time data led us to prototype a process which would permit measurement of response times during the course of a NetBench run. While the tools and methods are a crude hack, the project was successful enough to produce useful results [5], and the tools were further refined and used with a subsequent NetBench run on a larger, more interesting configuration [6].

Since NetApp does not have source code for NetBench, we could not enhance it with the time-stamping features of SPEC SFS. Even with the source, we might not have been able to do so—SPEC SFS is a synthetic benchmark [7] which generates the desired sequence of NFS requests, but NetBench is an application-level benchmark, generating relatively high-level file system calls. Not only doesn't a single call necessarily have the one-to-one correspondence with an SMB (a single CIFS transaction), as would be needed to measure the response time of a single operation, it might not even be generating SMB calls at all if, for example, it were being used to evaluate a PC NFS product.

Instead of modifying the benchmark itself, we captured network traffic between the server and a typical client. The resulting packet trace was subsequently analyzed off-line, matching client requests with the corresponding response packet(s) from the server and then computing the response time. The result does not reflect the time spent in the client's network protocol stack, unlike SPEC SFS, but the results can be meaningfully compared for two different CIFS file servers.

[1] CIFS (Common Internet File System) is simply a rechristened SMB (Server Message Block), to the confusion of many. Individual operations in the protocol are still commonly referred to as SMBs.

[2] NetBench, the most widely cited PC-oriented benchmark, is not actually specific to any underlying protocol—it can be used with any protocol a PC (or Mac) client can use, and thus can just as easily be used to evaluate NetWare, NFS, or even local file systems.

An additional benefit of this approach is that response time can be measured for any CIFS traffic, so other benchmark suites such as BAPCo's SYSmark for File Servers [8] could be substituted. (The response time tools might need to be enhanced to understand SMBs not encountered in the NetBench runs, but the hooks are in place to make this a relatively simple process.)

2. Data Collection

The test networks used were reasonably standard ones for NetBench, with client (load-generator) machines distributed evenly across multiple 100Base-TX networks. The key modification was the addition of a machine to capture packets flowing between a selected client and the server. In the early runs, this machine was on a hub with the server. This allowed us to select a different client on the fly in case the one initially being monitored dropped out of test. This flexibility did not prove to be necessary, and the packet capture machine was moved to a hub with the client to be monitored, as shown in Figure 1. This permitted the use of a dedicate switch port (or ports) to the server, a more typical configuration for a large installation.

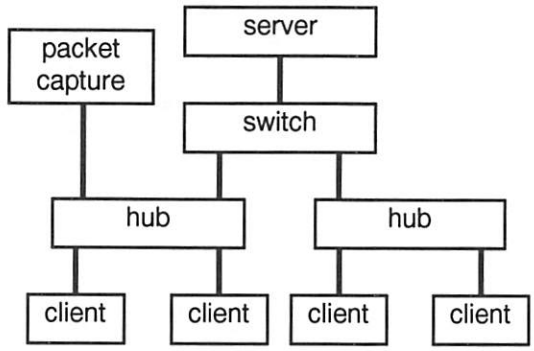


Figure 1: Switch-based benchmark configuration with packet capture machine on client hub

The large number of clients used in a NetBench run (our largest configuration used 200 clients, plus a few spares) makes a fully-switched network—with even the clients on dedicated switch ports—an unlikely scenario. However, this setup may be desirable for other benchmarks. It poses a problem since no shared networks are available on which to place the packet capture machine. Fortunately, many switches allow a monitoring port to be configured to which packets from one or more other ports are directed.

During the course of a NetBench run, the packet capture is started immediately after the client starts running the mix (one data point of the benchmark) and stopped shortly before completion. This is an exceedingly tedious process, but no hooks which can be used to automatically trigger and terminate the packet capture are apparent within NetBench. The packets missed at the beginning and end of this process are not a great concern since a large sample is still obtained.

3. Packet Capture Tools

In the early stages of this effort, Cinco Networks' NetXRay software [9] was used to perform the packet capture. It ran on a spare PC in our lab, and could decode SMB traffic, "printing" the decoded trace to a file. This reduced the time required to develop the Perl script used to analyze the data, but not the execution time of the analysis. When we started working with large samples, we found that the printer drivers were intolerably slow, and NetXRay occasionally died while attempting to decode some packets. We ended up writing our own decoding tool, but NetXRay's decoding served our prototyping process well.

A discussion in the SPECweb mailing list led us to suspect the resolution of the times reported by NetXRay [10]. An NT version of the SPECweb96 benchmark had been released, but it was discovered that the NT timer resolution was "only tens of milliseconds at best." Several Ultra SPARCs were in our lab as SPEC SFS load generators. The `snoop` utility which comes with Solaris 2 looked promising, with a claimed accuracy of 4 microseconds. We decided to switch to this packet capture tool for future runs, which required some modifications to the SMB decoder to accommodate the different format for the capture files.

While `snoop` gave us better timer resolution, we found that it also tended to drop packets, often in large quantity. A sample large enough to provide statistically meaningful results was still obtainable, but reworking the analysis code to properly match up SMB commands and their responses without being tripped up by dropped packets was a challenge.

The moral of this is that during benchmark runs for which accurate timing is important, a dedicated packet capture device such as a Network General Sniffer [11] is probably a worthwhile investment, instead of trying to make a general-purpose computer perform this specialized job well. (One would have thought that since

Network Appliance promotes the value of dedicated, appliance-like devices for specific tasks, we would have realized this sooner!)

4. Packet Trace Analysis

The core of the response time measurement is an analysis tool which studies the captured packet traces, looking for SMB commands and the corresponding replies from the server. The time-stamps on the captured packets are used to compute the time from when the first part of the command appeared on the network until the last packet of the response appeared. This tool was written in Perl for convenience of initial implementation and modification. It's quite slow and uses a prodigious amount of memory. Rewriting it in C would speed it up immensely, and perhaps reduce the memory demands, but it isn't run that often and we can obtain the resources when needed, and thus have not felt sufficient need to justify the investment in a full rewrite of the code.

The analysis is broken into three passes. The first pass scans the input, doing basic lexical analysis and sanity checking, then converting the data into an internal format. TCP sequence numbers are studied to catch packets retransmitted by TCP because they were not acknowledged by the receiver, and, with packet length information, to detect when packets have been dropped by the packet capture process. Packets which are continuations of an SMB command or response which required multiple packets are linked back to the initial packet.

The second pass ensures that at least the first and last packet of a multiple packet command or response are present. (Intermediate packets aren't important since they don't influence the overall response time of the operation.) More error checking is done and various counts are updated.

The third analysis pass examines packets which are the initial packet of an SMB command. The command is matched to a response based on Multiplex ID, with TCP sequence numbers and other consistency checks used to ensure that a response is indeed the one which corresponds to the command. (Multiplex IDs recycle, and protracted packet drops can lead to unfortunate coincidences if one is not careful.) If all the checks pass and the last packet of the command is present, the response time is computed by subtracting the time-stamp on the initial packet of the command from the final packet of the response and then tabulated.

5. Special Handling of Write_Raw SMBs

Determining a meaningful response time for the Write_Raw SMB poses an interesting challenge, as if the many artifacts of the data collection process where not challenge enough. This SMB begins with the client sending a relatively small chunk of initial data along with a reservation request for a much larger (up to 65,535 bytes) block of data. The server saves the initial data, reserves space for the large block of data, and responds to the client. The client then sends the block of data.

In the most frequently observed case, the server does *not* generate any response to acknowledge the receipt of the block of data. Any error is reported to the client in the next access to the file handle. This access can be arbitrarily far in the future, so there is no way to measure the response time for this portion of a Write_Raw command.

Without any direct acknowledgment of these SMBs, there is no way to measure response time for these operations in their entirety from a passive, external observation. (A TCP acknowledgment is generated, perhaps as part of another response, but this has no relationship to when the command actually completed.) These operations are therefore reported in two parts, the initial portion, including its response time, and the second, asynchronous portion.

6. Response Time Reporting

Appendix A is an example report, taken from the 80-client data point of a NetBench run against a Network Appliance F630 filer (file server appliance) running an early version of Data ONTAP 4.1. This is just one of seven reports from one NetBench run—obviously a more compact presentation is desirable for easy comparison and reference.

SPEC SFS combines response times into one number for each data point. We were not comfortable with the idea of collapsing the data that much for our work, both because we weren't sure if doing so would be meaningful, and because we were trying to compare two different *dialects* of the CIFS protocol—NetApp software does not yet implement the NT dialect, which has some extensions that offer significant performance benefits. We chose to sort the SMBs observed in the data into four groups of similar operations (detailed in Table 1) and to report response times for each group. Subsequent analysis was based on these groupings.

Read	Read+X	
Write	Flush_File	
	Write_Bytes	
	Write_Raw	
	Write+X	
Open/Close	Close_File	
	NT_Create+X	
	Open+X	
Other	Check_Directory	Get_File_Attributes
	Create_Directory	Get_File_Attributes2
	Delete_File	Get_Ext_Attribute
	Find_Close2	Get_Ext_Attrbute
	Rename_File	Set_File_Attributes
	Delete_Directory	Set_File_Attributes2
	Transaction2 - Find_First	Transaction2 - Get_FS_Info
	Transaction2 - Find_First2	Transaction2 - Get_Path_Info
	Transaction2 - Find_Next	Transaction2 - Get_File_Info
	Locking+X	Transaction2 - Set_File_Info

Table 1: Categories of SMB operations

Minimum, maximum, and median response time are included in the report, along with the average response time. Except during debugging, the average has been the most interesting statistic.

7. Comparing Two Real Servers

Figure 2 shows the throughput results from NetBench against a NetApp F630 and a Compaq ProLiant 5000 with hardware RAID running Windows NT. The system configurations are summarized in Table 2; further details along with the complete benchmark results are in [6].

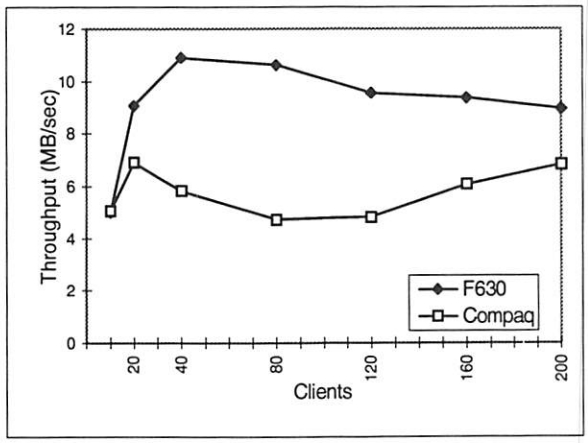


Figure 2: NetBench results for 10-200 clients

Server	NetApp F630	Compaq ProLiant 5000
Software	Data ONTAP 4.1 beta	Windows NT 4.0 with SP3
Processor	Alpha 21164A	Pentium Pro with 512KB L2 cache
CPUs	1	4
Speed	500 MHz	200 MHz
Memory	512MB plus 32MB NVRAM	1GB
SCSI	built-in (2 port)	2 SMART-2
Disks	26 F/W SCSI Seagate Barracuda	28 F/W SCSI Seagate Barracuda
RAID	RAID-4	2 hardware RAID-5, striped
Network	4 100Base-TX	4 100Base-TX

Table 2: Server Configurations

The drop-off in throughput of NT beyond 20 clients was expected—previous tests had shown that the performance of the Compaq suffered greatly once the working set exceeded memory size [5].

What was surprising was the *improvement* in throughput for the Compaq once the working set exceeded the server's memory. This caused considerable consternation for several weeks, until the response time analysis software could be reworked to handle the `snoop` output and the many packet drops it contained.

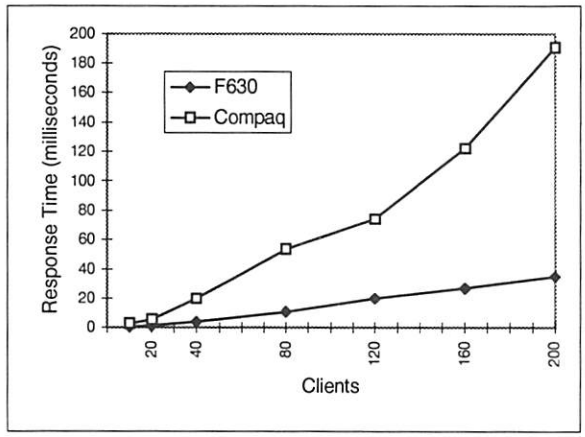


Figure 3: Response times for *Read* SMB group

The response time data finally helped explain this seemingly bizarre behavior. Figure 3 shows the response time for the *Read* group of SMBs for both of the servers tested. (Response times for the other SMB groups in Table 1 are presented in [6].) When the Compaq's throughput began to increase, the slope of the response time curve increases significantly—a steep price for the increased throughput. One plausible explanation is that NT switches algorithms as the load increases, from one tuned to provide clients with the best performance to one tuned for the convenience of the server. (Anecdotal evidence suggests that this sort of non-linear behavior is not unprecedented in NT benchmarking.)

The Network Appliance F630, in contrast to the Compaq running NT, degraded gracefully under increased load. Response time began low and stayed relatively low, an expected (and intended) benefit of NetApp's micro-kernel software architecture [12,13].

8. Conclusions

Studying file server response time is an interesting and enlightening exercise. Ideally, response time measurement is built into the benchmark. Lacking that, passively monitoring network traffic while running an existing benchmark and analyzing the packet traces off-line, despite being a crude hack, can produce useful response time data.

9. Availability

To encourage further studies in this area, the tools described in this paper will be made available in the free software section of Network Appliance's web site, http://www.netapp.com/technology/free.html. Please send any enhancements or fixes to this software to the author at kls@netapp.com.

References

1. Andy Watson, Bruce Nelson, "LADDIS: A Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark," *Proceedings of the 6th USENIX Large Installation System Administration Conference (LISA VI)*, pp. 33-38, Long Beach, California, October 1992.
2. Mark Wittle, Bruce E. Keith, "LADDIS: The Next Generation In NFS File Server Benchmarking," *Proceedings of the 1993 Summer USENIX Technical Conference*, pp. 111-128, Cincinnati, June 1993.
3. *Microsoft Networks SMB File Sharing Protocol (Document Version 6.0p)*, Microsoft Corporation, Redmond, Washington.
4. ZDBOp—NetBench, *http://www.zdnet.com/zdbop/netbench/netbench.html*, Ziff-Davis Publishing Company.
5. Karl L. Swartz, Andy Watson, *CIFS Filer Performance Measured with NetBench (TR-3015)*, Network Appliance, Santa Clara, California, 1997.
6. Karl L. Swartz, Andy Watson, *F630 Filer Performance Measured with NetBench (TR-3019)*, Network Appliance, Santa Clara, California, 1997.
7. Andy Watson, *NFS Performance with NetApp Filers (TR-3008)*, Appendix A, Network Appliance, Mountain View, March 1996.
8. *SYSmark for File Servers*, http://www.bapco.com/sysfs.htm, BAPCo, Santa Clara, California.
9. *Cinco Networks*, http://www.cinco.com/.
10. SPECweb private distribution list, January-February 1997.
11. Sniffer Network Analyzer, http://www.ngc.com/product_info/sna/sna_dir.html, Network General Corp., 1997.
12. Dave Hitz, *An NFS File Server Appliance (TR-3001)*, Network Appliance, Mountain View, California, January 1997.
13. Andy Watson, *Multiprotocol Data Access: NFS, CIFS, and HTTP (TR-3014)*, Network Appliance, Mountain View, California, December 1996.

Appendix A: Response Time Analysis Report for NetApp F630 (80 clients)

```
**********************
*  f630 - 80 clients  *
**********************

Packets:          91748
   IP/TCP:        74112    80.8%
   IP/TCP Ack:    17623    19.2%
   IP/TCP UAk:       13     0.0%

   retrans:            6     0.0%
   dropped:         7065  (or more)

Continue:         26787    29.2%

Commands:         44510    48.5%

   SMB:           23665    53.2%
   continue:      12614    28.3%
      sync:        5907           46.8%
      async:       6707           53.2%
   TCP ack:        8219    18.5%
   TCP retry:        13     0.0%

Elapsed time:   000:10:51.79540 (651.79540 seconds)
```

SMB Command	Count	% Tot	Avg. Pkts	Fastest	Slowest	Median	Average	StdDev
Check_Directory	235	1.0%	2	0.00032	0.03820	0.00724	0.00890	0.0049
Close_File	1412	6.0%	2	0.00021	0.04957	0.00729	0.00869	0.0049
Delete_File	269	1.1%	2.0	0.00096	1.64518	0.00811	0.04184	0.1945
Find_Close2	474	2.0%	2	0.00016	0.03768	0.00722	0.00869	0.0047
Flush_File	65	0.3%	2.0	0.00234	0.02620	0.00707	0.00865	0.0046
Get_File_Attributes	2062	8.7%	2.0	0.00026	0.15250	0.00753	0.00914	0.0055
Get_File_Attributes2	1264	5.3%	2	0.00034	0.04445	0.00686	0.00777	0.0043
Locking+X	289	1.2%	2	0.00048	0.03927	0.00702	0.00842	0.0046
Open+X	1863	7.9%	2	0.00029	0.04770	0.00751	0.00911	0.0047
Read+X	8735	36.9%	3.6	0.00020	0.17699	0.00732	0.01063	0.0126
Rename_File	41	0.2%	2	0.00143	0.02945	0.00771	0.00969	0.0049
Set_File_Attributes	1	0.0%	2	0.01390	0.01390	0.01390	0.01390	
Transaction2	820	3.5%	2.0	0.00032	0.09482	0.00779	0.00967	0.0060
Find_First	469	2.0%	2	0.00073	0.09482	0.00755	0.00943	0.0063
Find_Next	266	1.1%	2	0.00032	0.04912	0.00848	0.01000	0.0055
Get_FS_Info	85	0.4%	2	0.00390	0.03716	0.00792	0.00997	0.0056
Write_Bytes	5716	24.2%	2.8	0.00034	0.77066	0.00774	0.01010	0.0179
Write_Raw	261	1.1%	7.1	0.00699	0.20911	0.02325	0.02672	0.0157
async portion			25.5					
Groups:								
Open/Close	3275	13.8%	2.0	0.00021	0.04957	0.00739	0.00893	0.0048
Other	5455	23.1%	2	0.00016	1.64518	0.00727	0.01043	0.0441
Attributes	3412	14.4%	2	0.00026	0.15250	0.00719	0.00866	0.0052
Directory	1754	7.4%	2	0.00016	1.64518	0.00751	0.01422	0.0773
Locking	289	1.2%	2.0	0.00048	0.03927	0.00702	0.00842	0.0046
Read	8735	36.9%	3.6	0.00020	0.17699	0.00732	0.01063	0.0126
Write	6042	25.5%	3.0	0.00034	0.77066	0.00789	0.01080	0.0181

```
Response time distributions per command:

   Check_Directory:
        <0.001        4     1.7%     1.7%
        <0.01       155    66.0%    67.7%  **********************
        <0.1         76    32.3%   100.0%  **********
```

```
Close_File:
     <0.001       31    2.2%    2.2%
     <0.01       981   69.5%   71.7%  **********************
     <0.1        400   28.3%  100.0%  *********

Delete_File:
     <0.001        1    0.4%    0.4%
     <0.01       159   59.1%   59.5%  ******************
     <0.1        101   37.5%   97.0%  ************
     <1            4    1.5%   98.5%
     <10           4    1.5%  100.0%

Find_Close2:
     <0.001        8    1.7%    1.7%
     <0.01       334   70.5%   72.2%  ************************
     <0.1        132   27.8%  100.0%  *********

Flush_File:
     <0.01        51   78.5%   78.5%  ***************************
     <0.1         14   21.5%  100.0%  *******

Get_File_Attributes:
     <0.001       25    1.2%    1.2%
     <0.01      1310   63.5%   64.7%  *********************
     <0.1        726   35.2%  100.0%  ***********
     <1            1    0.0%  100.0%

Get_File_Attributes2:
     <0.001       35    2.8%    2.8%
     <0.01       980   77.5%   80.3%  *************************
     <0.1        249   19.7%  100.0%  ******

Locking+X:
     <0.001        4    1.4%    1.4%
     <0.01       219   75.8%   77.2%  *************************
     <0.1         66   22.8%  100.0%  *******

Open+X:
     <0.001       15    0.8%    0.8%
     <0.01      1225   65.8%   66.6%  *********************
     <0.1        623   33.4%  100.0%  ***********

Read+X:
     <0.001      105    1.2%    1.2%
     <0.01      6031   69.0%   70.2%  ************************
     <0.1       2562   29.3%   99.6%  *********
     <1           37    0.4%  100.0%

Rename_File:
     <0.01        26   63.4%   63.4%  *********************
     <0.1         15   36.6%  100.0%  ************

Set_File_Attributes:
     <0.1          1  100.0%  100.0%  ********************************

Transaction2:
     <0.001        3    0.4%    0.4%
     <0.01       519   63.3%   63.7%  *********************
     <0.1        298   36.3%  100.0%  ************

Transaction2 (Find_First):
     <0.001        1    0.2%    0.2%
     <0.01       320   68.2%   68.4%  **********************
     <0.1        148   31.6%  100.0%  **********

Transaction2 (Find_Next):
     <0.001        2    0.8%    0.8%
     <0.01       146   54.9%   55.6%  ******************
     <0.1        118   44.4%  100.0%  **************
```

```
    Transaction2 (Get_FS_Info):
         <0.01        53   62.4%   62.4%  ********************
         <0.1         32   37.6%  100.0%  ************

    Write_Bytes:
         <0.001       54    0.9%    0.9%
         <0.01      3702   64.8%   65.7%  ********************
         <0.1       1954   34.2%   99.9%  ***********
         <1            6    0.1%  100.0%

    Write_Raw (synchronous portion only):
         <0.01         3    1.1%    1.1%
         <0.1        257   98.5%   99.6%  ********************************
         <1            1    0.4%  100.0%

    Group - Open/Close:
         <0.001       46    1.4%    1.4%
         <0.01      2206   67.4%   68.8%  **********************
         <0.1       1023   31.2%  100.0%  **********

    Group - Other:
         <0.001       80    1.5%    1.5%
         <0.01      3702   67.9%   69.3%  **********************
         <0.1       1664   30.5%   99.8%  **********
         <1            5    0.1%   99.9%
         <10           4    0.1%  100.0%

    Group - Other.Attributes:
         <0.001       60    1.8%    1.8%
         <0.01      2343   68.7%   70.4%  **********************
         <0.1       1008   29.5%  100.0%  *********
         <1            1    0.0%  100.0%

    Group - Other.Directory:
         <0.001       16    0.9%    0.9%
         <0.01      1140   65.0%   65.9%  **********************
         <0.1        590   33.6%   99.5%  ***********
         <1            4    0.2%   99.8%
         <10           4    0.2%  100.0%

    Group - Other.Locking:
         <0.001        4    1.4%    1.4%
         <0.01       219   75.8%   77.2%  *************************
         <0.1         66   22.8%  100.0%  *******

    Group - Read:
         <0.001      105    1.2%    1.2%
         <0.01      6031   69.0%   70.2%  ************************
         <0.1       2562   29.3%   99.6%  *********
         <1           37    0.4%  100.0%

    Group - Write:
         <0.001       54    0.9%    0.9%
         <0.01      3756   62.2%   63.1%  ********************
         <0.1       2225   36.8%   99.9%  ************
         <1            7    0.1%  100.0%

Warnings:
   final packet of SMB missing:                          24    0.0%
   final packet of response missing:                     21    0.0%
   preceding packet(s) dropped:                        7065    7.7%
   response MID mismatch; looking ahead:                137    0.1%
   response MID not matched after 4 attempts:           137    0.1%
   retransmitted or out-of-order packet:                  6    0.0%
```

Brazos: A Third Generation DSM System[‡]

Evan Speight and John K. Bennett

Department of Electrical and Computer Engineering
Rice University
Houston, TX 77005
{espeight,jkb}@rice.edu

Abstract

Brazos is a third generation distributed shared memory (DSM) system designed for x86 machines running Microsoft Windows NT 4.0. Brazos is unique among existing systems in its use of selective multicast, a software-only implementation of scope consistency, and several adaptive runtime performance tuning mechanisms. The Brazos runtime system is multithreaded, allowing the overlap of computation with the long communication latencies typically associated with software DSM systems. Brazos also supports multithreaded user-code execution, allowing programs to take advantage of the local tightly-coupled shared memory available on multiprocessor PC servers, while transparently interacting with remote "virtual" shared memory. Brazos currently runs on a cluster of Compaq Proliant 1500 multiprocessor servers connected by a 100 Mbps FastEthernet. This paper describes the Brazos design and implementation, and compares its performance running five scientific applications to the performance of Solaris and Windows NT implementations of the TreadMarks DSM system running on the same hardware.

1 Introduction

Recent improvements in commodity general-purpose networks and processors have made networks of multiprocessor PC workstations an inexpensive alternative to large bus-based distributed multiprocessor systems. However, applications for such distributed systems are difficult to develop due to the need to explicitly send and receive data between machines. By providing an abstraction of globally shared memory on top of the physically distributed memories present on networked workstations, it is possible to combine the programming advantages of shared memory and the cost advantages of distributed memory. These distributed shared memory (DSM) runtime systems transparently intercept user accesses to remote memory and translate these accesses into messages appropriate to the underlying communication media. The programmer is thus given the illusion of a large global address space encompassing all available memory, eliminating the task of explicitly moving data between processes located on separate machines.

Both hardware DSM systems (e.g., Alewife [19], DASH [17], FLASH [15]) and software DSM systems (e.g., Ivy [18], Munin [6], TreadMarks [14]) have been implemented. Because of the traditionally higher performance achievable on engineering workstations relative to personal computers (PCs), the majority of existing DSM systems are Unix-based. Recent increases in PC performance, the exceptionally low cost of PCs relative to that of engineering workstations, and the introduction of advanced PC operating systems combine to make networks of PCs an attractive alternative for large scientific computations.

Software DSM systems use page-based memory protection hardware and the low-level message passing facilities of the host operating system to implement the necessary shared memory abstractions. The large size of the unit of sharing (a page) and the high latency associated with accessing remote memory combine to challenge the performance potential of software DSM systems[18]. A variety of techniques have been developed over the last decade to address this challenge [6, 10, 14]. DSM systems built using these techniques can be roughly grouped into three "generations": early systems like Ivy [18] that employ a sequentially consistent memory model [16] in a single-processor

[‡] This research was supported in part by substantial equipment donations from Compaq Computer Corporation and Schlumberger Company, and by the Texas Advanced Technology Program under Grant No. 003604-016. John Bennett was on sabbatical leave at the University of Washington while a portion of this work was conducted.

workstation environment, second generation systems like Munin [5] and TreadMarks [13] that employ a relaxed memory consistency model on similar hardware, and third generation systems that utilize relaxed consistency models and multithreading on a network of multiprocessor computers.

This paper describes the design and preliminary performance of Brazos, a third generation DSM system that executes on x86 multiprocessor workstations running Windows NT 4.0. Brazos is unique among existing DSM systems in its use of selective multicast, software scope consistency, and adaptive runtime performance tuning. Brazos uses selective multicast to reduce the number of consistency-related messages, and to efficiently implement its version of scope consistency [11]. Brazos uses a software-only implementation of scope consistency to reduce the number of consistency-related messages, and to reduce the effects of false sharing[2]. Finally, Brazos incorporates adaptive runtime mechanisms that ameliorate the adverse effects of multicast by dynamically reducing the size of pages' "copysets"[6], as well as an early update mechanism that improves overall network throughput. Brazos provides thread, synchronization, and data sharing facilities like those found in other shared memory parallel programming systems.

Brazos has been designed to take advantage of several Windows NT features, including true pre-emptive multithreading; support for the TCP/IP transport protocol, and in particular, multicast support; and OS support for symmetric multiprocessing (SMP) machines. Unlike most previous DSM systems, the Brazos runtime system is itself multithreaded. This allows computation to be overlapped with the long communication latencies typically associated with software DSM systems. Brazos also supports multithreaded user-code execution, allowing programs to take advantage of the local tightly-coupled shared memory available on SMP PC servers, while transparently interacting with remote "virtual" shared memory physically resident on other clusters. Brazos consists of four parts: a user-level library of parallel programming primitives, a service that allows the remote execution of DSM processes similar to the Unix **rexec** service, a memory management device driver that allows two virtual addresses to be mapped to the same physical address, and a Windows-based graphical user interface. Brazos currently runs on a cluster of Compaq Proliant 1500 multiprocessors connected by FastEthernet. In addition to describing the design and implementation of Brazos, this paper compares the performance of Brazos to the performance of Solaris and Windows NT implementations of the TreadMarks DSM system[14] running on the same hardware platform. For the applications studied, Brazos offers superior performance to both implementations of TreadMarks.

The rest of this paper is organized as follows. Section 2 describes some of the important differences between Unix and Windows NT as they relate to the development of software DSM systems. Section 3 discusses the design issues related to the Brazos system, and motivates many of the implementation decisions. Section 4 presents some preliminary performance results of Brazos relative to the TreadMarks DSM system, which represents the existing state-of-the-art in software-only DSM systems. Section 5 provides a brief discussion of related work. Section 6 concludes the paper and describes plans for future development.

2 Unix/Windows NT Differences

Windows NT differs substantially from Unix. The major differences that directly affect DSM implementation and performance include:

- Windows NT has native multithreading support build into the OS.
- BSD-style signals are not available.
- Exception handling is implemented through structured exception handling.
- Windows NT implements TCP/IP through the WinSock user-level library.

2.1 Multithreading

One of the significant features of the Windows NT operating system is the native support for multithreaded operation. Windows NT provides support for multiple lightweight threads executing within the same process address space. The Win32 API provides a rich set of calls to address threading issues, including support for thread priority manipulation, synchronization, thread context manipulation, and thread suspension and resumption. Standard Unix does not provide for lightweight threads, although there are several lightweight thread packages, such as Pthreads, that are available to run on top of Unix.

2.2 Signals

Unix makes use of *signals* to inform the operating system of events that must be handled. Signals that are used extensively in software DSM systems include SIGIO, which indicates that an I/O operation

is possible on a file descriptor (a *socket*), and SIGALARM, used as a timing device. The SIGIO signal is generally used to notify a Unix DSM system that an asynchronous message from another process is waiting to be received. An asynchronous message on any socket causes the function associated with the SIGIO signal to be invoked immediately, unless the user has explicitly blocked the delivery of the signal. The **select()** function must then be used to determine which socket has available data.

Windows NT does not support this kind of upcall mechanism. Instead, when a socket is made asynchronous in Windows NT, the function that will handle the asynchronous message is also specified. Thus, the function to be called is tied to the *socket* instance instead of a signal, allowing asynchronous messages to invoke different handler routines depending on the socket on which they are received. Additionally, the receipt of an asynchronous message does not automatically halt other threads (i.e., user-code threads). This allows independent threads to process incoming messages concurrently, while still allowing computation to proceed within scheduling guidelines.

2.3 Structured Exception Handling

Coherence is maintained in a page-based DSM system by setting page protection attributes to indicate the access permissions for a specific shared page. Processes "invalidate" pages in memory by altering the page protection such that any attempt to read or write to the page causes an access violation fault. For example, the Munin [5] and TreadMarks [13] DSM systems install interrupt handlers that specify a function to be called when a page-access violation occurs. When an access to an invalid page occurs, the DSM system enters the interrupt handler, and messages are sent to other processes to acquire the data needed to bring the page up-to-date in order to allow the user program to continue.

Windows NT accomplishes exception handling through a mechanism known as *structured exception handling* (SEH). SEH allows a greater amount of control over how exceptions are handled. Instead of installing a separate handler for each exception, as is done in Unix, Windows NT implements SEH through the **try-except** block. The **try-except** block is similar in flavor to the **throw-catch** block in C++, but it is used to trap software or hardware generated exceptions. The routine identified by the **__except()** keyword is called whenever any exception is raised within the **try-except** block, and this routine decides whether to handle the exception, pass the exception up to the operating system for handling, or continue execution without addressing the exception. In this way, exceptions that occur in different parts of code can be easily handled in different ways.

The code fragment below shows the use of the **try-except** block in the Brazos DSM system. The function **UserMain()** is the entry-point to the user code, and is called by each user thread. By placing the **try-except** block around this function, Brazos catches any page-access violations caused by threads accessing invalid pages. The function **AccessViolationHandler()** is the Brazos equivalent of the interrupt handler function installed in Unix-based DSM systems.

```
__try {
     UserMain(GlobalId,LocalId);
__except(AccessViolationHandler());
```

Table 1 gives the measured performance for two virtual memory operations (running on the same hardware) for both Windows NT and Solaris, a Unix System V derivative available from Sun Microsystems. The two systems are comparable in speed for setting the protection attributes of a page. However, Windows NT is more than twice as fast as Solaris handling an access violation, due to the lower overhead of the **try-except** block relative to the interrupt handling capabilities of Solaris. In practice, only those applications that exhibit a large number of access violation faults will substantially benefit from this difference.

OS	Page Protect	try-except *or* Segv Handler
Win NT 4.0	7.0 µsec	20 µsec
Solaris 2.5.1	6.57 µsec	47 µsec

Table 1. System Call Timings

2.4 WinSock

Processes in Brazos use functions in the WinSock Programming API to communicate with other processes in the system. All WinSock API calls are implemented in the WinSock library. With BSD sockets, some calls are direct system calls into the operating system, while other calls are made to functions in a static library that is linked at compile-time. Consequently, there is more overhead associated with many of the WinSock calls than with

their BSD counterparts, which can result in higher per-message overheads and lower overall throughput.

Figure 1 shows the average network throughput achievable between two machines connected by 100 Mbps FastEthernet. The graph shows throughput for Windows NT/WinSock as well as Solaris. These tests were conducted on the same hardware to rule out any variation due to architectural differences. The test conducted was a simple request-reply sequence that typifies the type of communication pattern seen in DSM systems. The client sends a 20 byte request to the server, which responds with a message of a length specified in the request. This loop is repeated 20,000 times, and the average results for different response sizes are presented here. The gray area shows the range of response sizes that would be expected in most DSM applications (from a few bytes up to a single 4 Kbyte page).

Figure 1 shows that for all response sizes in the typical DSM operating range, Solaris achieves a higher throughput than Windows NT, although neither system attains even half of the possible throughput until the response size exceeds 4 Kbytes. For responses of 16 Kbytes and 32 Kbytes, WinSock is able to stream data out more quickly, after paying the cost of initiating the message. Additionally, UDP messages can be up to 64 Kbytes in length under WinSock, but are limited to 32 Kbytes under Solaris.

3 Design of Brazos

Brazos has been designed to take advantage of the features available to the Windows NT and WinSock programmer. In particular, Brazos makes use of multithreading to overlap communication with computation; multicast to reduce the number of messages sent across the network; scope consistency [11] to reduce false sharing; and adaptive runtime support to implement an early update protocol and a page migration facility.

3.1 Design Overview

The Brazos user-level library is statically linked with user applications at compile-time and provides the interface between user code and DSM code. The most important part of the Brazos library is the interface for capturing accesses to invalid data and initiating messages with other processes in the system. This is accomplished by placing a **try-except** pair around the user code (see Section 2.3). The Brazos API also includes synchronization primitives in the form of locks and barriers, which can be used to provide synchronization both between threads in different processes as well as between threads in the same process. Routines for error reporting, statistics gathering, and data output are also provided in the Brazos API.

Windows NT does not ship with a method of starting a process on a remote machine similar to the Unix **rexecd** daemon. Therefore, Brazos includes a *service*

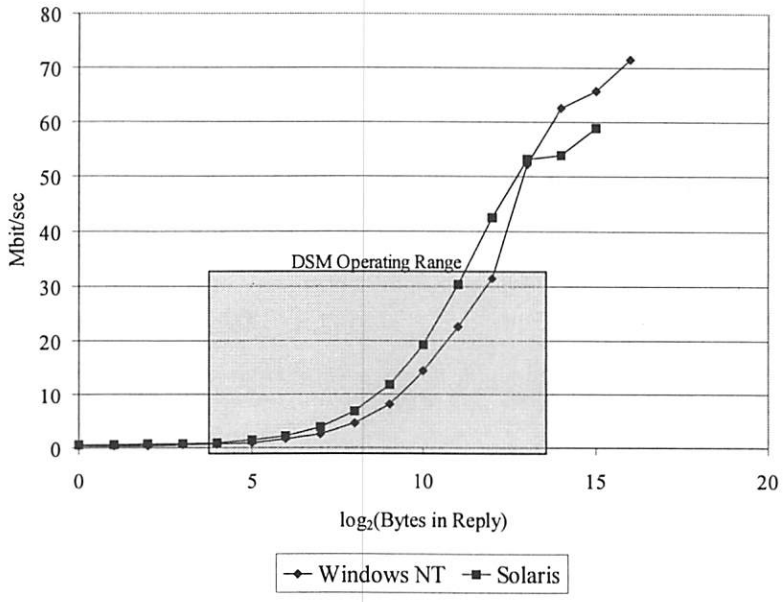


Figure 1. Average Network Throughput

that must be installed on each machine that will run Brazos code. This service listens for incoming DSM session requests, authenticates encrypted passwords that users must have to run a Brazos session, starts and manages current DSM sessions running on the local machine, and provides a mechanism for the owner to remotely kill a runaway DSM application.

In a multithreaded DSM system such as Brazos it is necessary to have a mechanism to allow the DSM system threads to update a page of shared memory without changing a page's protection. If a DSM system thread changes a page's protection in order to bring it up to date, there is a chance that a user thread will read part of the page before the updating is complete. There are two solutions to this problem. The first is to suspend all user threads when the system needs to atomically change the contents of a shared page. This method carries a high cost, especially when the number of user threads per process is large. The second method is to provide a mechanism to map two virtual addresses to the same physical page in memory, as Unix provides with the **mmap()** call. This allows there to be two different sets of protections associated with a single physical page. Windows NT does not provide a call similar to **mmap()**, but we altered the **mapmem** device driver in the Windows NT Device Developers Kit to provide this functionality.

Brazos includes a graphical user interface that provides mechanisms for specifying the number of user threads to start in each process, the mapping of threads to processors, the priority at which the DSM application should run, extensive statistics gathering and presentation mechanisms, and a service that can probe the network for hosts willing to accept a DSM session. Brazos can also be run in console-application mode when a Windows desktop is not available (i.e., during a telnet login session).

3.2 Multithreading

The Brazos DSM system utilizes multithreading at both the user level and DSM system level. Multiple user-level threads allow applications to take advantage of SMP servers by using all available processors for computation. Coherence is maintained between threads on the same machine through the available hardware coherence mechanisms. In addition to user-level multithreading, the Brazos runtime system itself is multithreaded. There are two main system threads in Brazos. One thread is responsible for quickly responding to asynchronous requests for data from other processes and runs at the highest possible priority. The second thread handles replies to requests previously sent by the process. As a practical matter, it is necessary for any DSM system written for Windows NT to be multithreaded to some degree, because it is difficult to interrupt a thread that is executing computationally intensive user code to cause it to respond to an asynchronous request for data. This is due to the lack of the signal-style upcall mechanism described in Section 2.2. This multithreaded aspect of Brazos allows a greater amount of computation to communication overlap, especially if there are more processors located on a given server than the number of user threads assigned to it. Finally, the use of a separate thread to handle incoming replies allows Brazos to maintain multiple simultaneous outstanding network requests, which can significantly improve performance [22].

3.3 Software Scope Consistency

DSM systems must maintain data consistency to ensure that threads do not access stale or out-of-date data that was written by a thread on another machine. Although a detailed discussion of the many consistency models used in shared memory systems is beyond the scope of this paper, we will briefly outline the major consistency protocols in use in modern DSM systems.

Sequential consistency (**SC**) [16] is the most intuitive, but also most restrictive, consistency model. Sequential consistency requires that all data accesses be consistent with a global ordering that does not violate program order. The simplest method of implementing sequential consistency requires threads to globally invalidate a page after every write to a shared variable on that page. This guarantees that no two threads will access out-of-date data, but can result in unacceptably high communication overhead in software DSM systems [5].

To reduce the amount of communication, consistency constraints can be relaxed by guaranteeing that shared data is only up-to-date after specific synchronization operations have been performed. For example, *release consistency* (**RC**) [8] guarantees that data is current only after a thread has performed a release operation. Simply put, a release operation can be thought of as the releasing of a lock variable or the departure from a barrier. Between release operations, it is the user's responsibility to ensure that no two threads perform competing accesses to the same storage space in memory (competing accesses are multiple accesses, one of which is a write). By relaxing the consistency model in this way, software DSM systems can buffer writes until they are required to be globally performed by the semantics of the consistency protocol. This results in substantial performance benefits because of the large reduction

in communication overhead [5]. *Lazy release consistency* (**LRC**) [13] further delays the propagation of invalidations until a synchronization variable is next acquired.

Scope consistency (**ScC**) [11], introduced as an enhancement to the SHRIMP AURC system [3], is a relaxed consistency model that seeks to reduce the *false sharing* present in page-based DSM systems. False sharing occurs when two or more threads modify different parts of the same page of data, but do not actually share the same data element. This leads to unnecessary network traffic, and can be a significant performance problem for DSM systems due to the large granularity of sharing. Scope consistency divides the execution of a program into *global* and *local scopes,* and only data modified within a single scope is guaranteed to be coherent at the end of that scope. Global scope delimiters include global synchronization events such as barriers. After a global scope is closed (completed), all shared data in the program is guaranteed to be coherent. A lock acquire-release operation is an example of a *local scope*. When a thread acquires a lock, it enters a new local scope. All changes made until the closing of the local scope (i.e., the lock release) are guaranteed to be visible to the next acquirer of the lock, but not changes made before the lock acquisition. This is in direct contrast to **RC**, which guarantees that all shared data is coherent after a release, regardless of when the shared write occurred or what type of release was performed.

The Brazos implementation of scope consistency (**SScS**) differs from that described in [11] in two ways: **SScS** is a software-only implementation of scope consistency that requires no additional hardware support, and Brazos uses **SScS** in conjunction with a distributed page management protocol similar to TreadMarks[14] as opposed to a home-based system such as Munin [5] and AURC [3]. In distributed page-based protocols, each process maintains dirty portions of each shared page of data, requiring processes to communicate with all other processes that have a modified portion in order to bring an invalid page up to date. In a home-based system, however, processes flush changes to a designated "home process", which always has the most up to date copy of a page. Other processes simply send a single message to the home process to re-acquire an invalid page. Brazos uses a distributed page management algorithm because for systems without DSM hardware support, distributed page management protocols outperform home-based page management protocols [13].

In order to be more precise about the specific conditions required to implement scope consistency for local scopes, the following conditions delineate the differences between release consistency and scope consistency using release-consistency nomenclature[‡]. Conditions associated with scope consistency only are shown in **[boldface]**. Conditions associated with both release consistency and scope consistency are shown in normal type. The use of the term "performed with respect to" in these conditions is consistent with that of [11].

1. Before an ordinary load or store is allowed to perform with respect to any other processor, all previous acquires must be performed.
2. Before a release is allowed to perform with respect to any other processor, all previous ordinary loads and stores **[after the last acquire to the same location as the release]** must be performed **[with respect to that processor]**.
3. Synchronization accesses (acquires and releases) must be sequentially consistent with one another.

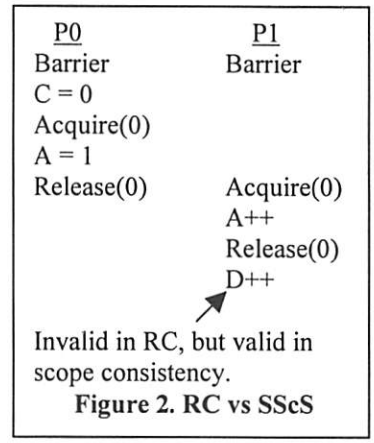


Figure 2. RC vs SScS

Figure 2 demonstrates these concepts. In Figure 2, assume that variable **A** is on one page of shared memory, and variables **C** and **D** are on another. In a coherence protocol such as release consistency, variable **D** will be invalid in process **P1** after the **Release(0)** performed by process **P0**. This is because variable **C** was written to by process **P0** before the release, and variables **C** and **D** are on the same shared page. Under scope consistency, variable **D** will not be invalidated because the write to **C** by process **P0** occurred outside of the local scope delimited by the

[‡] In order to use terminology consistent with previous work in the area of relaxed consistency, "opening" and "closing" a *local* scope, as defined in [11], will be considered to be equivalent to an "acquire" and "release" to the same location. "Opening" and "closing" a *global* scope, as defined in [11], will be considered to be equivalent to consecutive barrier events.

Acquire(0)-Release(0) pair. The effect of the write to the page containing variables **C** and **D** will not be propagated to process **P1** until the end of the current global scope. Had the programmer wanted to ensure that the correct value of **C** would be available to process **P1** after the critical section performed by process **P0**, either the write of **C** should be moved into the critical section, or a global scope must be used after the write to variable **C**.

In Brazos, **SScS** provides two main benefits. First, in the code fragment just discussed, the new value of **A** written by process **P0** is sent along with the lock grant to process **P1**, thereby eliminating a message that would result from the fault of **P1** when trying to increment **A** under **RC**. Secondly, the page containing **C** and **D** is falsely shared in Figure 2.

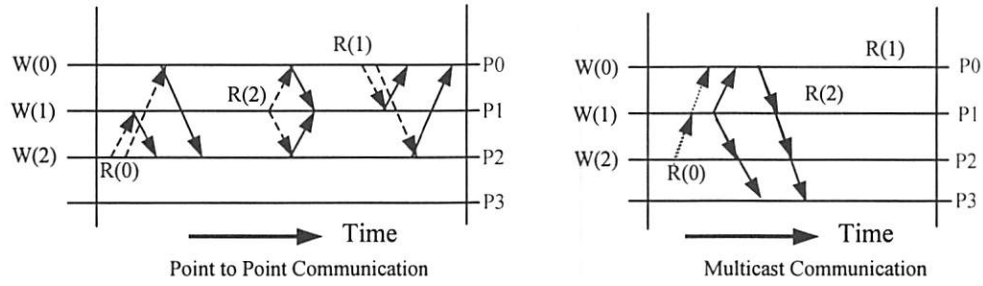


Figure 3. Point to Point vs. Multicast Communication

Therefore, process **P1** will find the page invalid under **RC**, even though **P0** did not actually modify **D**. **SScS** removes the effects of this false sharing by not invalidating the page containing **C** and **D** when the lock ownership is transferred.

In some situations, programmers may be faced with situations where it is not easy to switch from **RC** semantics to **SScS** semantics. For such instances, Brazos provides both a release consistent lock release primitive, which will flush all invalidation messages before allowing the release to complete; and a lazy release consistent acquire primitive that flushes updates to modified pages to all processes at a lock acquire. The flexibility provided by the inclusion of these two extra lock primitives makes porting existing parallel programs to Brazos easier.

3.4 Multicast Communication

In order to reduce the number of consistency-related messages, and to efficiently implement scope consistency in software, Brazos makes use of the multicast primitives provided by the WinSock 2.0 library [23]. In a time-multiplexed network environment such as Ethernet, sending a multicast message is no more expensive than sending a point-to-point message, and large reductions in both the number of messages sent and the number of bytes transferred to maintain coherence can be achieved by specifying multiple recipients for each message. Brazos uses multicast to reduce consistency-related communication traffic during *global* synchronization as follows.

When a process arrives at a global synchronization point (i.e., a barrier), the process sends a message to a statically assigned barrier manager indicating that the process has arrived at the barrier. Included in this message is a list of pages that the process has written to since the last synchronization point (*dirty* pages). When the barrier manager receives notification from all processes, the manager collates information regarding dirty pages, and sends a message to each process indicating the pages that should be invalidated, who has dirty pieces of the page, and that the process is free to proceed from the barrier.

After being released from the barrier, threads will begin faulting on pages that were invalidated by the barrier manager. Since the manager also communicated which processes have copies of all dirty pages, a single multicast message may be sent to the subset of all processes that have dirty pieces of the page. These processes, in turn, multicast their response to not only the requesting process, but to all processes in the current copyset for that page. The responses come in the form of *diffs*, which are runlength encodings of the changes made to the page since the last invalidation. Thus, processes that need the *diffs,* but have not yet faulted on the pages, will receive *indirect diffs* for these pages, with the intent of bringing them up-to-date before a page fault resulting from the accessing of the invalid page occurs.

This mechanism is illustrated in Figure 3, which shows the differences in communication patterns between a distributed page-based DSM system that relies on point-to-point communication and Brazos. In Figure 3, processes **P0**, **P1**, and **P2** each write to a different variable on the same page of shared data sometime before the first barrier, **B1**, as indicated by **W(0)**, **W(1)**, and **W(2)**. After **B1**, the processes each read a value written by another process, as indicated by **R(0)**, **R(1)**, and **R(2)**. The messages required to satisfy each of these read requests are shown with arrows, with request messages using dashed lines and response messages using solid lines. As can be seen, it takes 12 point-to-point messages to completely bring processes **P0**, **P1**, and **P2** up-to-date for the data written before **B1**, but only 3 messages are required for the method employing multicast. However, in the multicast implementation, process **P3** also receives indirect *diffs* for the page, even though these are not used. This potential performance problem is addressed in the next section.

3.5 Adaptive Runtime Support

Adaptive performance tuning mechanisms can have a beneficial effect on performance when used to tailor runtime data management to observed behavior[1, 2]. Brazos employs four adaptive techniques: dynamic copyset reduction, early updates, an adaptive page management protocol, and a performance history mechanism.

3.5.1 Dynamic Copyset Reduction

One disadvantage with multicast is the potential harmful effect of unused indirect *diffs*, as in the case of **P3** in Figure 3. Receiving multicast *diffs* for inactive pages does not increase network traffic. However, it does cause processors to be interrupted frequently to process incoming multicast messages that will not be accessed before the next time that the page is invalidated, detracting from user-code computation time. The dynamic copyset reduction mechanism ameliorates this effect by allowing processes to drop out of the copyset for a particular page, causing them to be excluded from multicast messages providing *diffs* for the page. The decision to drop out of the copyset is made by counting the number of unused multicast *diffs* received for a specific page. When this number reaches a certain threshold, the process will place this page on the "to be dropped" list, and this list of pages is piggybacked on the next barrier arrival message. The process then removes itself from the current copyset. When a thread in the process next faults on the page, the entire page is retrieved from the manager and any outstanding *diffs* from other processes are retrieved and applied immediately. This adaptation is particularly beneficial to performance in situations where two processes actively share a page of data, and neither of these processes is the page manager. Although Brazos follows a distributed page management algorithm (i.e., processes must retrieve modified pieces of a dirty page from several processes, not just one), each shared page is assigned a page manager at the beginning of execution from which the page is retrieved by other processes only on the first access to the page. Because the page manager is always in the current copyset for a shared page, the manager will always receive indirect multicast *diffs* for the page. The adaptive copyset reduction mechanism allows the manager to drop out of the copyset and migrate the manager status to a process actually involved in the sharing. This reduces the number of useless indirect *diffs* that the original page manager receives.

3.5.2 Early Updates

Another form of runtime support provided by Brazos is an *early update* mechanism. Because the majority of the DSM-related network activity in release-consistent DSM systems occurs immediately after synchronization events, network traffic in these systems tends to be bursty. Referring again to Figure 3, assume that **R(0)**, **R(1)**, and **R(2)** all happened immediately after the barrier. This would result in processes being sent indirect multicast *diffs* for pages for which the process currently has outstanding requests, resulting in extraneous messages. In order to reduce this burstiness, processes in Brazos note pages for which they receive indirect *diffs* while they are waiting for a response to a request for *diffs* to the same page. At the next global scope, processes send the page numbers of these early update pages to the barrier manager along with the barrier arrival message. The manager distributes the list of the new early update pages with the barrier release message to all processes, and thereafter processes multicast their changes for all early update pages in a single bulk transfer message before each arrival at a barrier. This eliminates the flurry of network traffic resulting from threads simultaneously faulting on the same page in memory immediately after a synchronization point, since the early update pages will not be invalidated after the barrier. Pages may switch back from the early update protocol to the default multicast invalidation protocol by the dynamic copyset reduction mechanism described above. Specifically, processes count how many updates go unused for each page. When this number reaches a certain threshold, the process drops from the copyset. When

the number of processes in the copyset reaches one, the page's protocol is changed back from early update to the multicast invalidate protocol.

3.5.3 Other Techniques

Brazos incorporates two other adaptive techniques. The first of these allows pages to be managed with either a home-based protocol similar to Munin [5], or a distributed page protocol similar to TreadMarks [13]. The Brazos runtime system adaptively alters pages' management protocol based upon observed behavior in order to provide the best management technique for each shared page.

Brazos also incorporates a history mechanism that allows the runtime system to more quickly adapt to programs' behavior. Brazos saves information regarding the performance of the adaptive protocols in a file for each application. These files store information about how well the various adaptive techniques worked for each program variable. This low level of granularity is desirable because the mapping between pages and data may change across program execution, but program variables generally do not. The history mechanism and dynamic page management protocol were not used in obtaining the results presented in Section 4. Details on these techniques can be found in [22].

3.6 Brazos Program Development

Users write Brazos programs using familiar shared-memory programming semantics. Any shared data in the system may be transparently accessed by any thread without regard to where in the system the most current value for that data resides. The Brazos runtime system is responsible for intercepting accesses to stale data and bringing shared pages up-to-date before program execution is allowed to continue.

Programs written for Brazos specify the function **UserMain()** instead of the normal **main()** function as the entry point into user code. User code is linked with the static library **brazos.lib** at compile-time. This library contains the DSM system code for maintaining shared-memory across the network, providing synchronization between threads (both within the same process and between processes) and collecting statistics on the performance of the local DSM process. The resulting console-based executable is started on each machine participating in the DSM run through the use of the Windows NT service described in Section 3.1.

The Brazos DSM programming library provides parallel programming macros based on a superset of the PARMACS macro suite [4]. Locks and barriers are the two forms of synchronization available to the parallel programmer, and the synchronization macros for these may be used without regard to where the synchronizing threads are located (e.g., in the same process, or between processes). Windows NT synchronization primitives are embedded inside the PARMACS synchronization macros to allow for this transparency, which also allows the same executable to be run as a DSM program across servers, or as a strictly hardware-based shared memory program on a single SMP server. All shared data must be dynamically allocated through the PARMACS macro **G_MALLOC** in order to make the DSM subsystem aware of which portions of the address space must be maintained as shared data. All other data is considered to be private to the threads running in each individual process.

4 Performance of Brazos

This section presents preliminary performance results for various configurations of eight processors. Figure 4 presents speedup numbers comparing Brazos to two versions of the TreadMarks DSM system: TMK-SOL is the standard release of TreadMarks 1.0 on Solaris, and TMK-NT uses a version of TreadMarks 1.0 that we have ported to Windows NT. All data were obtained on a network of four Compaq Proliant 1500 servers connected by a 100 Mbps FastEthernet. Each Proliant 1500 has two 200 MHz Pentium Pro processors with 192 Mbytes of main memory.

Application	Input Set
SOR	2048 X 2048 matrix
ILINK	*Amish* input set
Barnes Hut	32,768 bodies
Water	729 molecules, 10 steps
Raytrace	*balls4* input set

Table 2. Application Input Sets

Five applications were studied. SOR is a nearest-neighbor algorithm used to solve differential equations and was taken from the TreadMarks sample application suite. ILINK [9] is a parallel implementation of a genetic linkage program that traces genes through family genealogies. Barnes Hut solves hierarchical n-body problems and was taken from the SPLASH benchmark suite[21]. Water calculates forces and potentials in a system of water molecules, and was also taken from the SPLASH suite. Finally, Raytrace is a graphics rendering

application from the SPLASH-2 benchmark suite [24]. Input data for each of these programs is shown in Table 2.

The speedups shown in Figure 4 are relative to the uniprocessor execution time for the same operating system, e.g., the speedups for TMK-SOL are shown relative to the uniprocessor execution time under Solaris, and the Windows NT configurations are shown relative to the uniprocessor execution times under Window NT. The uniprocessor times are not the same for the TMK-NT and TMK-SOL implementations due to compiler differences. For further details, see [22].

SOR achieves a higher speedup under both Windows NT configurations than under Solaris. Because SOR has little communication, the higher achievable network throughput for Solaris shown in Figure 1 does not give the Solaris implementation a significant advantage. The overlap of communication with available computation allows TMK-NT to slightly outperform TMK-SOL. The Brazos implementation further improves performance by making use of the available native hardware shared memory support for threads located on the same machine, reducing the overall network communication by half. SOR's sharing is all pair-wise between at most two threads, therefore the use of multicast does not help the performance of the Brazos implementation.

Examining the performance of ILINK, we see that the TreadMarks implementation under Solaris performs 9% better than the Windows NT implementation. ILINK has a moderate amount of communication, and the increased network throughput of Solaris give TMK-SOL an advantage over TMK-NT for this application. However, the use of the available native hardware shared memory support again gives the Brazos configuration a significant advantage in ILINK. Additionally, the sharing patterns in ILINK favor the use of multicast, and these two factors decrease the overall communication needed by 238% over the course of ILINK's execution. Finally, the early update adaptive mechanism further reduces communication by another 37%, leading to the 54% performance improvement of ILINK under Brazos.

Barnes Hut displays behavior similar to that of ILINK, with Brazos significantly outperforming either TreadMarks implementations. Barnes Hut benefits slightly from multithreading, but the use of multicast reduces communication rates by more than seven-fold through a reduction in the effects of false sharing. Barnes Hut displays a large amount of false sharing, with threads faulting on pages that are write shared, even though individual data elements on the pages are not shared. In Brazos, processes receive indirect *diffs* for falsely shared data before the access violation occurs, eliminating the detrimental effects

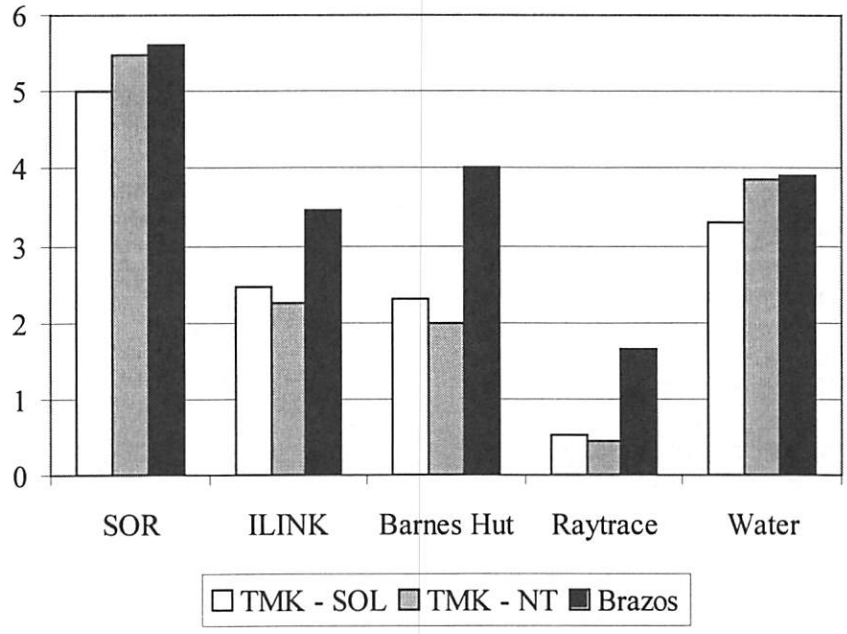


Figure 4. Performance of Brazos vs TreadMarks on 8 Processors

of the false sharing, reducing communication, and leading to a near doubling in performance for Barnes Hut.

Raytrace has been shown in previous studies to benefit from the use of scope consistency [11]. With the hardware currently used by Brazos (very fast processors and a relatively slow communication media), programs like Raytrace that have large amounts of communication do not achieve good performance. Raytrace was included in this study to show the effects of scope consistency in Brazos, and to demonstrate that an all software implementation of scope consistency based on a distributed page management algorithm can be beneficial to applications that exhibit the correct sharing patterns. Because all shared data in Raytrace that must be propagated between threads is contained within small critical sections, Brazos is able to reduce the number of messages sent by 202% through a reduction in false sharing. This reduction leads to a 614% decrease in the number of bytes sent for Raytrace by not invalidating pages that are written outside of these critical sections. As a result, Brazos obtains a speedup of 1.64 on 8 processors, whereas both TreadMarks implementations are so communication bound that a slowdown in moving from 1 to 8 processors was observed.

Although Water performs better under both Windows NT implementations, it benefits minimally from the use of multicast and user multithreading. Consequently, Water does not perform appreciably better under Brazos than TMK-NT. This is mainly because over 70% of the messages in Water are synchronization messages, which currently do not benefit from the use of multicast. More detailed results on these and other application programs can be found in [22].

5 Related Work

We are aware of only one other software DSM system built using Windows NT [12]. The Millipede project provides a simple programming interface and portability, while implementing adaptive measures such as thread migration and load balancing. Brazos builds upon ideas from several earlier DSM systems, including Ivy [18], Munin [6], and TreadMarks [13]. To the best of our knowledge, Amoeba [20] is the only other DSM system to make use of group communication (multicast), although recent industry interest (most notably the IP Multicast Initiative from Stardust Technologies) in the use of multicast may make the use of multicast more widespread.

The implementation of DSM on SMP machines has been addressed in several systems. In [10], a multiple writer protocol with automatic updates is described. This design relies on specific hardware extensions to implement automatic update, whereas Brazos uses commodity PC's and networks. Erlichson et al. [7] describe a single-writer, epoch-based release consistency DSM design for SMP machines. They conclude that network bandwidth limited the performance potential of this approach. We have shown that network traffic can be reduced significantly through the use of multicast and adaptive protocols.

Scope consistency was first introduced in [11] and relied on specific hardware support provided in the SHRIMP multicomputer [3] to achieve performance gains over a software-only implementation of LRC. Software-only scope consistency models have been proposed [25], but these systems are home-based systems rather than a distributed page-based system like Brazos.

6 Conclusions and Future Work

This paper has described Brazos, a software DSM system that runs under Windows NT 4.0 on a network of PC servers. We have demonstrated that such a system can be competitive with networks of Unix computers for scientific applications, despite the lower per-message overhead of Unix. This was accomplished by taking advantage of available local hardware coherence mechanisms, support for multithreading, a relaxed consistency model, and selective multicast. We have briefly presented these concepts, and have shown their aggregate effect on the performance of five scientific applications. We are working to improve the performance of the adaptive techniques presented here, as well as to develop new techniques. We are also investigating mechanisms for thread migration across distributed processes, thread checkpoint and restart, and support for multiprogramming. Finally, we are working on a faster transport protocol to reduce the high startup overhead we currently observe with WinSock.

Source code for Brazos, including future enhancements, will be made available for non-commercial use via the World Wide Web in the near future.

References

[1] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Write and Multiple Writer. In *The Third International Symposium on High-Performance Computer Architecture.* p. 261-271, 1997.

[2] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type - Specific Memory Coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming.* p. 168-176, 1990.

[3] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* p. 142-153, 1994.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. 1987: Holt, Rinehart and Winston, Inc.

[5] J.B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency.* Ph.D.Thesis, Rice University, Houston, 1993.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *Transactions on Computer Systems*, **13**(3):205-243,1995.

[7] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. Soft FLASH : Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Systems.* p. 210-220, 1996.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture.* p. 15-26, 1990.

[9] S.K. Gupta, A.A. Schaffer, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Integrating Parallelization Strategies for Linkage Analysis. *Computers and Biomedical Research*, **28**:116-139,1995.

[10] L. Iftode, C. Dubnicki, E.W. Felton, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture.* p. 14-25, 1996.

[11] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *The 8th Annual ACM Symposium on Parallel Algorithms and Architectures.* 1996.

[12] A. Itzkovitz, A. Schuster, and L. Wolfovich, *Millipede: Towards Standard Interface for Virtual Parallel Machines on Top of Distributed Environments*, Technical Report 9607, Technion IIT, 1996.

[13] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* Ph.D.Thesis, Rice University, 1995.

[14] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference.* p. 115-131, 1994.

[15] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* p. 302-313, 1994.

[16] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Distributed Multiprocess Programs. *IEEE Transactions on Computers*, **C-28**(9):690-691,1979.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, **25**(3):63-79,1992.

[18] K. Li. Ivy: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing.* p. 94-101, 1988.

[19] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D.Thesis, Yale University, 1986.

[20] S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R.V. Renesse, and H.V. Staveren. Amoeba - A Distributed Operating System for the 1990s. *IEEE Computer*, **23**(4):44-53,1990.

[21] J.P. Singh, W.-D. Weber, and A. Gupta, *SPLASH: Stanford Parallel Applications for Shared-Memory*, Technical Report CSL-TR-91-469, Stanford University, 1991.

[22] E. Speight. *Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors*. Ph.D. Thesis, Rice University, Houston, 1997.

[23] Stardust Technologies. *Windows Sockets 2 Application Programming Interface*. 1996.

[24] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* p. 24-36, 1995.

[25] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation.* p. 75-88, 1996.

Moving the Ensemble Communication System to NT and Wolfpack[†]

K. Birman W. Vogels K. Guo M. Hayden
T. Hickey R. Friedman R. van Renesse Al. Vaysburd

Dept. of Computer Science[‡]
Cornell University

S. Maffeis
Olsen & Associates

Abstract

Cornell University has developed a group communications and membership management tool, called the Ensemble system, which provides the basis for introducing guarantees such as reliability, high availability, fault-tolerance, consistency, security and real-time responsiveness into applications that run on clustered parallel computers or high speed networks. Ensemble tools are flexible, extremely transparent, and achieve high performance. Our development started under Unix in 1995, but by 1996 had enlarged to include NT as a primary target. This paper reviews Ensemble and then discusses the technical issues that arose when repositioning it to fit naturally and perform well under NT.

1 Introduction

Our research seeks to develop a new generation of groupware communication tools for modern networked environments, The Ensemble system offers support to the application developer who wishes to introduce guarantees such as reliability, high availability, fault-tolerance, consistency, security and real-time responsiveness into network applications. Although our project started on Unix platforms, a recent emphasis has been on the integration of the Ensemble tools into NT platforms and high-availability clusters of NT-servers such as those managed by Wolfpack.

The key to our work is to focus on what we call *process group communication structures* that arise in many real-world settings involving cluster-style computing, scaleable servers, groupware and conferencing, distributed systems management, and fault-tolerance. A process group is just a collection of processes running on some set of computers in a network. Our work uses process groups to support execution guarantees (for example, that a request to a critical server will be done even if a failure occurs, or will be completed within a real-time deadline), replicated data or computing, distributed coordination or control, and so forth. The emphasis of our effort is on the underlying communication systems support for this model, on simplifying and standardizing the interfaces within our support environment, and on making the model as transparent (hence, easy to use) as possible.

2 Isis, Horus, and Electra

Over the past 12 years, our project has developed a series of three related software systems. All solve the group communication problem, but they differ in focus. Our first was called the Isis Toolkit, and was developed at Cornell during the period 1985-1990. Isis was commercialized by Isis Distributed Systems (currently a division of Stratus Computer), and found success in settings like stock markets, air traffic control systems, electric power plant monitoring and control, factory-floor automation in VLSI fabrication plants, telecommunications systems, and banking/brokerage systems.

By 1990, we had encountered a number of potential Isis applications that pushed the system to its limits. These included very large-scale applications (with hundreds of participating computers) and cluster-styled systems, like the ones that Wolfpack is designed to support. Such systems often require more than the basic software fault-tolerance solutions offered by Isis. For example, many cluster-based systems require strong real-time

[†] Supported, in part, by DARPA ITO under ARPA/ONR contract N0014-96-1-10014, and in part by grants from Siemens, Intel and Microsoft.

[‡] Visit http://www.cs.cornell.edu/Info/Projects/Horus for information on how to contact the authors. Our software is available for general use at no fee; details on downloading and running it can be found on the web page.

guarantees as part of the fault-tolerance architecture. As an extreme case, a cluster computer used in an SS7 telephone switching system (an "IN coprocessor") must handle thousands or tens of thousands of call requests per second, guarantee 100ms response times for each event even as nodes fail and restart, and exhibit at most 3-secs downtime per year. Such requirements go beyond what Isis was able to provide.

We built the Horus system as a response to these needs. Developed over a 5-year period that started in 1991, Horus supports Isis-like functionality, but was designed to match its guarantees to the real needs of the application. We use this feature to tune Horus, so that it will give the best possible performance consistent with the reliability requirements of a specific use. The architecture is a modular one based on layered protocols. The protocol stack supporting a particular group communication application is "plugged together" at runtime. By hiding Horus behind standard interfaces, we obtained a good degree of transparency: this tactic allows us to slip group-based mechanisms into applications that were not designed with group communication as an explicit goal. Moreover, Horus supports an object-oriented interface to the CORBA architecture, which we call Electra.

Horus pushed well beyond the limits of Isis. For example, we used Horus to emulate an SS7 telephone switching coprocessor on a cluster of Unix systems. Our solution handled 20,000 telephone requests per second within the 100ms deadline, dropping less than 1% of calls (randomly) even if a failure occurred under peak load. For multicast applications over ATM networks, Horus achieved 85us end-to-end latency, and sustained throughputs of 85,000 1-byte multicasts per second.

As Horus matured, however, we encountered issues that lead to a complete reimplementation of the system. We are calling this most recent system *Ensemble*. In contrast to Horus and Isis, which were Unix-oriented, Ensemble is intended to run on NT as well as Unix and Linux, and is increasingly NT-centric in design. The body of the present paper focuses on issues raised by this shift in target platform.

3 The Ensemble System

Our key reason for moving to Ensemble concerns the method used to optimize Horus protocol stacks for good performance. It is well known that when protocols are built in a modular, layered manner, as in Unix streams or object-oriented communication architectures, the strict modularity of the layers can introduce substantial overhead. In Horus, we encountered this problem, and overcame it by developing a method for optimizing Horus protocol layers to drastically compress message headers and also shorten the critical path for most multicasts. When we applied these techniques to Horus, however, they increased the complexity of the system, reducing the benefits of modularity and flexibility.

Recoding Ensemble in OCaml, an object-oriented dialect of the ML language, made it possible to use a powerful theorem prover called Nuprl to do these sorts of transformations automatically. We are experimenting with using Nuprl to automatically generate highly optimized protocol stack that exhibits the low latencies and high throughputs that previously required a laborious hand-optimization. Although even compiled OCaml executes more slowly than C, the new protocols in Ensemble are currently faster than similar protocols in Horus, on the same hardware. Moreover, we are experimenting with using Nuprl to verify that Ensemble correctly implements critical protocols such as the ones that synchronize the reporting of group membership changes with the delivery of messages. Such formal proofs will be a valuable adjunct to exhaustive testing or other ad-hoc methods of convincing ourselves that these complex protocols work correctly. In the future, we may be able to gradually prove the correctness of more and more of Ensemble as a whole.

Users of Ensemble treat it as a collection of tools and communication primitives. Our users typically code applications in C, C++, Tcl/Tk, Java, SmallTalk or ML. They access Ensemble through "toolkits" that include libraries for replicating key portions of an application to make it fault-tolerant, replicating data for rapid access or parallelism, doing coordination and synchronization when multiple systems cooperate on a task, providing membership management for tracking the participants in an application, and automatically setting security keys. Ensemble is already integrated with Kerberos and PGP and we are now looking at support for other security architectures the RSA version of Kerberos, SSL, Fortezza, and IPSEC.

Like Horus, Ensemble can be configured at runtime to match the needs of the application and to exploit the properties of the network on which it is running. If desired, one application can run over several multicast groups simultaneously, configuring each to offer different properties. For example, a multimedia conferencing system could configure one multicast group for video, one for audio, and one for control. Each would employ

a protocol matched to the reliability and security needs for the kind of data it handles.

4 Porting Ensemble to NT.

Porting Ensemble to NT raised a number of issues, the first of which stems from our use of OCaml. OCaml is an object-oriented dialect of the ML language, and is structured to encapsulate portability issues in its runtime environment. OCaml can be used either as a platform independent bytecode interpreter, or a generator of native assembly code. Ocaml's support libraries are written in C, and the overall system exhibits only a slightly degraded performance when compared to implementations in to other high-performance languages.

Cornell researchers extended the OCaml runtime under Win32 with an interface to the communication subsystem (winsock). This proved difficult because of the need to retain compatibility with Unix. For example, WIN32 file and socket descriptors represent different objects and cannot be "mixed". In Unix, however, all types of file descriptors are considered to be of the same type. Thus, where Ocaml's runtime system previously could use a single *select* system call applied to all open file descriptors, the Win32 interface required a more complicated structure to treat the various file types separately but concurrently.

This said, the majority of the porting effort went into constructing a build environment that would work for Unix as well as Win32. The Win32 tools have different semantics or are severely crippled (nmake), but we decided not make use of ported versions of the Unix tools under NT to avoid a non-standard build process. The Ensemble source code is now maintained under Unix, but during the checkout process scripts are used to produce Win32 make and dependency files.

5 Matching the NT programming Model.

The de facto standard for programming open NT applications is the Component Object Model (COM), which was recently extended to take the distribution of objects into account (DCOM). Under NT, Ensemble provides its functionality through a collection of COM interfaces. Using these interfaces various levels of abstractions are exposed to the programmer, giving the developer full control at whatever level of detail is needed to implement the system under construction. Some of these are:

- *A low-level, event driven interface.* Internally Ensemble is a pure event driven system, and a subset of these events is exposed through a connectable COM interface. This allows higher level programming toolkits to be implemented using collections (groups) of COM objects.

- *A high level, object oriented interface.* We built a toolkit, called Maestro, which extends Ensemble with a collection of C++ classes that implement ADT's for essential types (endpoint ID's, messages, error handlers, etc), group-member/client-server abstractions and state-transfer for server objects. These types and their interfaces are exposed to COM programmers through a COM interface.

- *A highly available RPC interface.* Ensemble provides an RPC system that allows clients to access high-available objects in a transparent, fault-tolerant fashion. If the connection to an instance of an object fails, it will automatically reconnect the client to another instance of the object. Part of the RPC system is an inter-server protocol, which ensures that the servers are aware of the fail-over of clients between objects and are able to perform the necessary repair and garbage collection.

These three interfaces represent generic Ensemble services in the sense that they are available on all platforms that Ensemble runs on. The services are encapsulated in COM interfaces to ease the integration of this complex technology into the wold of everyday Win32 programmers.

We have found these interfaces successful in assuring a certain level of ease of integration, but they still require that the developer have some awareness of the distributed architecture of the system being developed. As such, these interfaces are normally used to implement high performance cluster-style servers that can provide load balancing, fault-tolerance, or parallel processing.

For clients accessing such servers, our intent is to offer a very high degree of transparency, so that the developer can work without being aware that the server is using Ensemble-based tools. We also interested in providing support for managing standard (Ensemble unaware) server objects in such a way that they can achieve the high reliability of a sort similar to that available for Ensemble aware server objects. This type of transparency is achieved through the use of Ensemble-aware proxy objects at the client nodes, through high-available referral objects that manage the access to server objects and through manipulation of the binding process at the OXID services.

For applications that use standard DCOM interfaces, it is possible to do completely transparent replication and management. Ensemble provides a *Distributed Object Container* that runs as an NT service at each available server node. Operations on the container server are automatically presented to the contained objects, which see identical invocation sequences (this is called "state machine replication"). By dropping a standard DCOM object into such a container, it can be replicated with no changes at all. In the future, we plan to extend the performance and functionality of our container by providing other replication options in addition to state machine replication, and by providing functionality for managing the group of objects as a whole. Eventually, we should be able to provide a "replication wizard."

Ensemble provides two additional services that increase the reliability of the overall distributed system: a high-availability directory service, accessible through COM interfaces as well through LDAP, and a replicated object store. The latter also provides highly-available storage for persistent object state. Regular COM objects can be initialized from this high-available storage using the familiar *IPersist** interface.

6 NT server clusters and Wolfpack

One of the areas for which Ensemble seems well matched is high availability cluster computing. The Ensemble tools can be used for cluster management (to achieve higher reliability both in availability and performance), and can also be used when developing applications that exploit the services of a high-availability cluster. Our research into cluster computing is looking at ways to combine the Ensemble tools with Wolfpack.

Wolfpack is a management system under development at Microsoft, which targets the market for highly available clusters of NT-servers. At present, Wolfpack provides generic application fail-over, whereby groups of applications are automatically restarted on an alternative node of the cluster in case of a failure of the first node. Selection of the alternative node is based on the (physical) resources that need to be present for the application to function. Clients using such a service experience a disconnect from the failed server, and must then reconnect to a new instance of the application running at the surviving node. Wolfpack assigns IP addresses that can migrate between the nodes with groups of cluster applications, enabling clients to use the same address to connect to a server independent of the node on which it is actually running.

To support fault handling by the restarted application Wolfpack provides disk sharing (at SCSI level) and registry replication between the two nodes. The new application instance is expected to read the state of the failed application instance from the shared disk and, through a recovery mechanism, to recover the state of the application at the time of a failure. In effect, an application that was running on a node when it fails will be restarted only on another node that offers an indistinguishable execution environment. This approach offers a practical means of supporting applications that are restartable but not cluster aware, and that cannot be actively replicated using process group software.

However, there are a number of drawbacks to this approach, relating primarily to scale and to achieving continuous availability instead of fail-over. For example, it would not have been possible to solve the telecommunications coprocessor problem using a disk-based state management approach, because recovery would vastly exceed the 100ms limit on latency for individual requests and the 3-second overall limit on downtime *per year* for such applications. Moreover, in this approach, adding nodes to the cluster does not increase the perceived reliability of an application, its availability (only a limited number of nodes can physically share a disk), or its performance (only one instance of the application can be active at the same time). In practice the scalability model presented by the current Wolfpack organization is that of islands of 2-node clusters.

Even on a 2-node system Ensemble can be used to provide nearly continuous availability without the need to physically share resources. This is done using a primary/backup configuration where the second application instance is used as a *hot standby*. Ensemble guarantees that the state of the second instance is in sync with the primary instance. Fail-over to the second instance can be achieved with the guarantee that no state has been lost and recover/rollback (with possible loss of transactions) is not needed. Ensemble also supports an alternative called full active replication, where the application is active on both nodes and provides continuous operational guarantees while exploiting the resources on both nodes. In the limit, Ensemble is able to generalize cluster and application management to achieve *n*-way fault-tolerance, where each node can be backed up by *n* other nodes.

N-way fault-tolerance offers the potential for load-balanced use of the available resources, and big wins through scalable parallelism and large in-memory cache or databases. Returning again to the telecommunications application, much of the complexity is associated with

keeping the right information in the database buffer pool (cache), because databases of customer profile information are too large to fit in the memory of a uniprocessor. A cluster can potentially support an arbitrarily large memory (simply by adding nodes), creating the potential for a completely memory-mapped solution. We believe that comparable requirements arise in many domains.

Integrating Ensemble closely with Wolfpack to augment its cluster management system presents challenges beyond the ones encountered in simply porting Ensemble to NT. The management API of Wolfpack provides what are called *clusters, groups* and *resources*, where a cluster manages a set of groups, which contain a set of resources. Failure detection is offered at the level of resources while migration policies apply to groups. In these respects Wolfpack goes beyond Ensemble, both by offering API's specialized to a problem that we have not focused upon in our work to date, and by supporting management functions extensible by third parties, that would typically be outside the scope of software developed in academia. Basically, Wolfpack supports the notion that a group of resources will automatically be "moved" from node to node if a node fails. Resources can be associated with preferred nodes, and hierarchies of dependencies can be created, so that resources will be restarted in the right order.

The best match for this organization within Ensemble arises in a tool that we call the *Maestro Group Manager*. The group manager is a service, built out of Ensemble components, which provides hierarchical group management by tracking the membership of a managed process or communication group and automatically reconfiguring a group when one of its members fails.

Wolfpack treats its membership management subsystem in a modular manner; hence it is possible to replace the standard membership module with a proxy component that has a tight interaction with the Maestro tool. Doing this would provide Wolfpack with a scalable node and process membership. In effect, Ensemble now functions as the subsystem responsible for tracking cluster and resource membership. However, this step also makes the basic Ensemble functionality available to Wolfpack developers. For example, parallel database query engines could exploit Ensemble's high speed data replication and synchronization support. Such mechanisms are needed in implementing basic database operations on parallel processors, and standard solutions would presumably appeal to database vendors. Secure applications could use Ensemble's secure replication mechanism to distribute and manage keys within cluster members. And the techniques we used to achieve real-time responsiveness in our telecommunications switch example would become available to Wolfpack application developers in domains that demand real-time performance guarantees. For example, we recently developed a high performance Web server with real-time response guarantees, using the same approach.

The functionality offered by the Ensemble tools is quite a bit more general than that currently used in Wolfpack or planned for the next releases. In particular, Ensemble can be used to manage a cluster, but can also be used as a groupware programming environment. In this case, Ensemble is accessed from the Java language combined with a Web browser plug-in or an Active/X control. The Wolfpack membership tracking and reconfiguration mechanisms, in contrast, are a module internal to Wolfpack and not exposed for purposes other than the ones just mentioned. We believe that for developers of applications in which groups of participants cooperate or coordinate (such as multi-user interactive games, virtual reality environments, or business applications involving conferencing and briefings with multiple participants), group communication tools are important, and also easy to use. We see big advantages to factoring out group membership tracking and communication functionality, so that a single subsystem can support these very varied potential uses.

7 Ensemble Roadmap

At the time of this writing, the NT version of Ensemble is stable, including the C++ and Java programming interfaces (our Unix versions have been stable for some time and we have a growing user community). Understanding how to fully embed group communication and multicast functionality into COM/DCOM, and thus Active/X, will take some time; we already knew how to do this on the Unix and CORBA side because we supported such an option for Horus. We expect that by the autumn of 1997, Ensemble will be easily useable from Java or C++ through several COM interfaces on NT platforms including Wolfpack, and that by late in the year, we will have a stable integration of Ensemble into DCOM. In the same time frame we will be designing a number of Active/X control interfaces, based on the Ensemble/COM interface, and targeting group collaboration and communication opportunities.

The longer-term vision of our effort revolves around the seamless, highly transparent, introduction of "strong properties" into network applications developed using standard tools and programming practices. A funda-

mental premise underlying our work is that most critical applications are being developed using conventional off-the-shelf building blocks and combined into applications using standard techniques. We have come to believe that even the most critical applications are essentially forced to do this because it represents the only practical way to take advantage of modern computing technology. The challenge, as we see it, is to "harden" such systems without requiring source-code changes.

We are also expanding our emphasis on security. We believe that the community seeking "high reliability" has a broad notion of what this should mean, in which security goals are at least as important as fault-tolerance. A major goal is to integrate Ensemble with all the prevailing security options for distributed computing, so that any Ensemble application can transparently obtain authentication credentials for the processes with which it interacts, can encrypt or sign sensitive data by simply specifying the need, and can obtain trustworthy services such as group membership management and routing.

Finally, we are developing a more flexible notion of reliability itself. In Isis, Horus and Ensemble, up to the present, reliability has tended to be of an all-or-nothing flavor. The ability to build flexible protocol stacks has opened the door to supporting other notions of quality of service within Ensemble, but we haven't taken very much advantage of this so far. A big goal of ours in 1997 is to begin to develop probabilistic protocol stacks, which might offer a way to trade off between reliability and other goals, such as steady latencies or scalability. As noted earlier, Isis began to run into performance issues as it scaled into the low hundreds of participants. We believe that Ensemble could scale to thousands or tens of thousands using protocols that substitute a rigorous notion of "very probable behavior" for the "guaranteed reliability" of the basic virtual synchrony model.

Success in this effort could have a broad impact on the reliability and security of mission-critical distributed computing systems. Today, a tremendous rollout of these systems is occurring in settings that include medical critical care, air traffic control, on-board avionics systems, financial systems, factory automation, and critical business applications. Such applications demand extremely high levels of reliability. By providing easily used reliability and security solutions, well integrated with standard platforms and programming tools, and flexible enough to match the properties provided to the needs of the user, these critical applications can be made safe and secure. Indeed, we look forward to the day when reliability and security tools will be a common feature of standard distributed operating systems, much like file systems, TCP/IP communication, and Web technologies.

Online information

http://www.cs.cornell.edu/Info/Projects/Horus

References

Building Reliable and Secure Network Applications. K. Birman, Manning Publishing Company (Greenwich, CT) and Prentice Hall, January 1997. 550pp.

Software for Reliable Networks. K. Birman and R. van Renesse. *Scientific American* 274:5 (May 1996), 64-69.

Horus: A Flexible Group Communications System. R. van Renesse, K. Birman and S. Maffeis. *Commun. of the ACM* 39:4 (Apr. 1996), 76-83.

Using Group Communication Technology to Implement a Reliable and Distributed IN Coprocessor. R. Friedman and K. Birman. *Proceedings of TINA '96: The Convergence of Telecommunications and Distributed Computing Technologies.* Heidelberg, Germany, Sept. 3-5 1996, 25-42. VDE-Verlag.

Software releases

The Horus project has produced two generations of software. The first generation, Horus, is stable, and available for general use. There are no licensing fees for research use of Horus. Horus commercial rights, however, have been exclusively licensed by Cornell University to Isis Distributed Systems, which is developing a commercial product in this area. Contact rcbc@isis.com (Dr. Robert Cooper) for details concerning the commercial product offering, support, or other services.

Cornell University is making Ensemble available at no fee in source form for both research and commercial researchers. The system is available today and additional releases are expected periodically during 1997 and 1998 as new functionality is completed.

Parallel Processing with Windows NT Networks[1]

Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ 85287-5406
Email: partha@asu.edu
http://cactus.eas.asu.edu/partha

Abstract

Workstation-based parallel processing is an area that is still dominated by Unix-based systems. We have been building new methods for shared-memory parallel processing systems on top of Windows NT based networks of machines.

As of present we have been involved in four related systems, called *Calypso NT, Chime, Malaxis* and *MILAN*. All of these are middleware, that is they are system level libraries and utility programs that allow programmers to utilize a network efficiently for high volume computations. Calypso was first built on Unix [BDK95], and then ported to Windows NT. Chime and Malaxis are NT systems and MILAN is still under the design phase.

This paper describes the systems, the techniques used to implement them on Windows NT and the roadblocks from a Unix programmer's point of view.

1. Introduction

This paper describes the experience of porting to and programming with Windows NT (from a Unix programmer's perspective) while implementing *four* related parallel processing projects on a network of computers. The paper also provides overview details about the parallel processing systems we have built. Our research efforts have produced system-level programs and libraries that run under Windows NT and allow parallel applications to utilize a network of workstations and have access to shared memory and provided fault tolerance. We have been working with Windows NT since January 1996 and have designed, implemented and ported, considerable amount of software. All the people involved with the projects had prior significant system development experience with Unix. Thus we started with our own set of *biases* and "*set-in*" ways. Therefore, switching to Windows NT has been fun at times and frustrating at others.

Two of the four projects under way have been implemented and tested. These are:

- *Calypso NT:* Calypso NT is a modification and port of a preceding Unix implementation. Calypso on Unix supports shared memory parallel programs that execute on a network of Sun machines and provides an efficient execution substrate coupled with low-cost fault tolerance and load balancing. The porting of Calypso to Calypso NT was our first experience with the Windows NT operating system. The Calypso NT system is being distributed for free and is available at *www.eas.asu.edu/~calypso*.
- *Chime:* Chime is an efficient, fault-tolerant implementation of the Compositional C++ parallel programming language on Windows NT. Some of the lessons learned from Calypso have migrated to Chime, but a lot of the inner architecture is different, due to a change in programming style from "Unix-ish" to "NT-ish".

Two newer NT-based projects are in progress. These are:

- *Malaxis:* An implementation of a Distributed Shared Memory (DSM) system for Windows NT. This system uses an innovative distributed locking mechanism that is coupled with barrier synchronization. This technique is expected to provide performance better than release consistent DSM systems. The actual performance has not been tested yet.
- *MILAN:* A metacomputing environment that extends the technologies prototyped in Calypso, Chime, Malaxis, and a Java-based system (Charlotte) [BKKW96], to provide a unified computing environment for robust general purpose computing on a networked platform.

Our current Windows NT platform consists of twelve Pentium-Pro-200 computers, four Pentium-133 computers and five Pentium-90 computers connected by a 100Mb/s Ethernet, running Windows NT 4.0. The development environment consists of Visual C++ 4.2.

[1] This research is partially supported by grants from DARPA/Rome Labs, NSF, and Intel Corporation.

2. From Unix to Windows NT

Like most academic research groups, we were a heavily Unix (SunOS) oriented group. Our research was hosted on Sun computers and all the participants were well versed in Unix and somewhat skeptical of Windows NT. So making the plunge into NT was venturing into uncharted waters.

As stated earlier, we started by attempting to port the Unix version of Calypso, to Windows NT. Since many of the mechanisms used by Calypso are operating system independent, we thought that porting would be a matter of replacing some Unix system calls with Windows NT system calls and recompiling. After a few months of attempts, using several GNU tools and libraries, it turned out that we were wrong.

We then discovered the key differences between Unix and NT. These differences cannot be shoved under the rug via user-level libraries, and affects the porting of system-level software. The differences include:

1. Windows NT does *not* support *signals*. There are a variety of mechanisms in NT for asynchronous events including threads, messages, events and so on, but they do not map on cleanly to signals.
2. Windows NT uses "Structured Exception Handling" (or SEH) which is quite different from what UNIX programmers are used to.
3. Windows NT does not provide a "remote shell" feature.
4. Threads are the mechanism of choice for handling any form of asynchrony - including those tasks normally done through signals in Unix.
5. Windows NT expects the applications to be integrated with the windowing system and preferably such applications should be developed with the MFC (Microsoft Foundation Classes) library.
6. Terminology is different, making things confusing and sometimes exasperating.

In spite of such differences, the similarities between Unix and NT are quite striking. Due to the functional similarities, differences are easy to overlook. Some good articles on porting strategies from Unix to NT exist in the documentation library that is bundled with the development environment (a.k.a. MSDN — Microsoft Developers Network).

3. Parallel Processing on Networks

In recent years, the focus of parallel processing technology has shifted from specialized multiprocessing hardware to distributed computing. The most important factor in the favor of distributed computing is that it can provide high performance at low cost. The computational power of a workstation cluster can surpass that of a supercomputer, if harnessed properly.[2] The advantages of using a network of workstations to implement a parallel processing system are evident from the development of a plethora of parallel processing solutions based on distributed platforms, in recent years.

These "*distributed*" parallel processing systems enable the programmer to exploit the hidden computational power of the networked workstations, but they do not always address many of the important issues. Most of these systems use their own programming models and/or programming languages that are not always easy to understand and require extensive modifications to existing software. Message passing systems, for instance, add a layer that facilitates data transfer and process synchronization using messages. Some parallel processing systems do not differentiate between the parallelism inherent in an application and the parallelism available at execution time. In such cases, the degree of parallelism is an argument to the program. However once the execution begins, the width becomes fixed. Therefore, issues such as failure recovery or appropriate distribution of the workload to account for slow and fast machines cannot be addressed elegantly.

3.1 Prior Work

Significant numbers of parallel processing systems have been built for use in networked environment. The notable ones are PVM, MPI and Linda.

The parallel processing systems can be loosely divided into two types, those that depend on a message passing scheme and those that use some form of global address spaces. Many systems provide message passing, or Remote Procedure Call facility built on top of a message passing. These include PVM [S90, GS92], GLU [JA91], Isis and so on. These systems provide a runtime library (and sometimes compiler support) to enable the writing of parallel programs as concurrently executable units.

Using global memory to make programs communicate has been established as a "natural" interface for parallel programming. Distributed systems do not support global memory in hardware, and hence, this feature has to be implemented in software. While systems built

[2] Effective and innovative harnessing of the computing power of workstation networks is one of the primary research goals of the MILAN project in general and Calypso in particular.

around Distributed Shared Memory (DSM) like IVY [Li88] Munin [DCZ90], TreadMarks [ACD+95] and Quarks [K96] and Clouds [DLA+90] provide a more natural programming model, they still suffer from the high cost of distributed synchronization and the inability to provide suitable fault tolerance. A mature system that uses a variant of the DSM concept is Linda [CG89]. Piranha [GJK93] provides a feature similar to Calypso in that is allows dynamic load sharing via the ability to add and subtract workers on the fly. However, the programming strategy is different, deleting workers need backing up tuples, and fault tolerance is not supported.

The issues of providing fault tolerance have generally been addressed separately from the issues of parallel processing. There have been three major mechanisms: *checkpointing, replication* and *process groups*. Such approaches have been implemented in CIRCUS [Coo85], LOCUS [PWC+81], and Clouds [DLA+90], Isis [BJ87], Fail-safe PVM [LFS93], FT-Linda [BS93], and Plinda [AS91] However, all these systems add significant overhead, even when there is no failure.

More recently several prominent projects have similar goals to us. These include the NOW [Pat+95] project, the HPC++ [MMB+94] project, The Cilk project [BL97] and the Dome [NAB+95] project. All these projects however use approaches that are somewhat conventional (RPC or message based systems with provisions for fault detection, checkpointing, and so on.)

While the majority of the systems run on Unix, there are a few systems that run on Windows NT. These include Win-PVM, Win32-MPI and Brazos [SB97].

4. The Calypso System

The design of Calypso [BDK95, DKR95] addresses efficient, reliable parallel processing in a clean and efficient manner. In particular the Calypso NT [MSD97] has the following salient features:

- **Ease of Programming:** The programmer writes programs in C or C++ and uses a language independent API (application programming interface) to express parallelism. The API is based on a shared-memory programming model which is small, elegant, simple and easy to learn.
- **Separation of Logical Parallelism from Physical Parallelism:** The parallelism expressed in an application, written using a high-level programming language, is logical parallelism. Logical parallelism is be kept separate from physical parallelism, which depends upon the number of workstations available at runtime.
- **Fault Tolerance:** The execution of parallel Calypso jobs is *resilient to failures*. Unlike other fault-tolerant systems, there is *no additional cost* associated with this feature in the absence of failures.[3]
- **Dynamic Load Balancing:** Calypso automatically distributes the workload among the available machines such that faster machines do more work compared to slower machines.
- **High Performance:** Our performance results indicate that the features listed above can be provided with minimal overhead and that a large class of coarse-grained computations can benefit from our system.

The core functionality of Calypso is provided by a unified set of mechanisms, called *eager scheduling*, collating *differential memory* and *Two-phase Idempotent Execution Strategy* (*TIES*). Eager scheduling provides the ability to dynamically exploit the computational power of a varying set of networked machines, that includes machines that are slow, loaded or have dynamically changing loading properties. The eager scheduling algorithm works by assigning tasks to free machines in a round robin-fashion until all the tasks are completed. The same task may be assigned to more than one machine (if all tasks have been assigned and some have not yet terminated). Consequently, free or faster machines end up doing more work than the machines that are slower or loaded heavily. This results in an automatically load balanced system. Secondly, if a machine fails (which can also be regarded as an infinitely slow machine), it does not affect the computation at all. Thirdly, computations do not wait or stall as a result of system's asynchrony or failures. Finally, an executing program can utilize any newly available machines at any time.

As it is obvious the memory updates in such a system need careful consideration, the remaining mechanisms ensure correct executions in spite of failures, and other problems related to asynchrony. To ensure that the inherent possibility of a multiplicity of executions due to eager scheduling results in exactly-once execution semantically, the TIES method is used. Furthermore, arbitrarily small update granularities in shared memory and the proper updates of memory are both supported by the collating differential memory mechanism.

The implementation of Calypso was first done on Unix. The Windows NT port preserves the Unix methodology

[3] This claim is well justified by our performance results. See section 8.

and replaces the signal handling and memory-faulting methods with NT specific handlers as described later.

5. The Chime System

Chime is an implementation of the shared memory part Compositional C++ [CK92] language on a network of Windows NT machines. The shared memory part of CC++ is designed for shared memory multiprocessors. It embodies many features that have been considered impossible if not difficult to implement on distributed machines. These include structured memory sharing (via use of cactus stacks), nested parallelism and inter-thread synchronization.

Chime implements these features of CC++, efficiently on a distributed system, making the distributed system look and feel like a real shared memory multiprocessor. In addition, Chime adds fault tolerance and load balancing.

A complete description of how Chime works is beyond the scope of this paper, but we will present some NT specific considerations.

The major difference between the implementation of Calypso and Chime is in the manner threads are used and contexts are migrated. Every site running Chime, runs two threads per process. The threads are called the "*Controlling*" thread and the "*Application*" thread. The controlling thread is responsible for all communication, memory-fault handling, thread context migration and scheduling. The application thread runs the code written by the programmer of the application.

As an example, consider the case when a application thread, running one of the parallel tasks of a parallel application decides to spawn a subtask, which is a nested parallel computation:

1. The application thread suspends itself and signals an event to the controlling thread.
2. The controlling thread saves the context of the application thread. This context will be used to create the parallel tasks on remote machines.
3. The controlling thread registers with a manager, the context of the application thread, the number of new tasks to be created, the stack of the application thread and the "continuation", i.e. the remainder of the application thread, that should be executed after the parallel tasks are over.
4. The manager then schedules the new threads on available machines.
5. A controlling thread on a worker machine picks up a task from the manager.
6. The controlling thread on the worker now crafts an application thread, with the same stack and context as the parent thread, and starts the thread.
7. Via some compiler tricks, the newly created thread executes the task it was supposed to execute.
8. Then, when the thread terminates, the updates it made to the global data and the stack are returned to the manager and its state is appropriately updated.

The above is just one aspect of the execution behavior implemented in Chime. The complete system supports proper scoping of variables, execution management, inter-thread synchronization and nested parallelism. The system consists of the Chime runtime library and a preprocessor that serve as a front end to the C++ compiler.

6. Malaxis and Milan

We are also building more systems using Windows NT. The two notable ones are (i) Malaxis, a DSM (Distributed Shared Memory) system that provides data locking and barrier synchronization (ii) MILAN, a metacomputing platform. Due to space constraints, the descriptions of these are omitted.

7. Using NT features

In order to implement software such as Calypso, and Chime, we needed some support from the operating system. The support included:

- Support for user level demand paging for implementing page-based distributed shared memory.
- Support for obtaining and setting thread contexts for implementing task migration and task scope preservation.
- Support for resetting the contents and the position of the user stack in order to implement distributed cactus stacks.
- Support for network communication.
- Support for asynchronous notification and exception handling.

It turned out that Windows NT supports all of these features—and in some ways more elegantly than Unix does. It was just necessary to expend considerable time and effort to work out the detail of usage and semantics.

Previously, when we were developing libraries for parallel computing in Unix we used the ubiquitous ASCII interface for all programs. The ASCII interface is *not* so ubiquitous under NT, in fact, it is thought of as arcane. NT programs that use textual interfacing are called "console applications". So far, except for some user

interfaces, most of out programs are console applications, though we intend to change this in the near future.

7.1 Memory Handling

Memory handling in Windows NT is different and in many ways superior to UNIX. NT has various states of memory allocation (reserved, allocated, committed, guarded and so on). These states allow (among other things) a set of threads to allocate address space and then later allocate memory when the need arises.[4]

Windows NT memory management is designed for use with threads and sounds like overkill to Unix programmers who are not heavy users of threads. We will discuss the threads issue in a later section. We found the functionality to be useful for allocating and protecting memory for supporting the dynamic distributed shared memory used by Calypso. The important memory management API functions are `VirtualAlloc` and `VirtualProtect`.

It is possible to protect any range of pages in memory against read or write or execute access. Access violation results in an exception, and the exception handler is provided with a plethora of information about the nature of the exception, something most Unix-based systems do not provide. Due to the tight coupling between the memory protection and exception handling, much of the Calypso memory system had to be reprogrammed, but the end result, we feel, is more elegant and extensible.

7.2 Exception handling

Exception handling was the major surprise. While UNIX uses signals, Windows NT uses *Structured Exception Handling (SEH).* SEH is *not* the same as C++ exception handling, which makes use of C++ keywords `throw`, `try` and `catch`. SEH uses the *`try-except`* construct, which allows programmer to specify a guarded scope to catch a hardware or software exception using the `_try` block. The `_except` block specifies the *exception handler* that may be executed based on the value returned by the *exception filter* at the time when an exception is raised. The mechanism is quite different from that of UNIX signals as the lexical structure of the program rather than one-time installation of the handler specify its activation. Once, the flow of control is out of the `_try` block, any raised exceptions can not be intercepted. While porting Calypso to NT, this restriction forced us to make some structural changes in the implementation.

In retrospect, there is nothing wrong in the NT approach. However for Unix programmers who think and breath `signals` the paradigm shift can be unnerving (it was for us). Also the lack of signals, at first made it look like doing things like process migration would be impossible. It is possible; it necessitates the use of threads.

7.3 Threads

Handling process migration and process context changes had us stumped for a bit. Without signals, a Unix programmer is lost! However, we soon discovered the power of threads under Windows NT.

Threads are one of Windows NT's strongest features. Its thread support is simple, elegant and works well. A thread is started by calling the `CreateThread` routine with a function as argument, the new thread executes the specified function. The threads are kernel scheduled and share all the global memory. Quite simple and intuitive. We have found threads to be quite useful, in many situations, specifically:

- Threads are very useful for process migration and implementing distributed stacks (next section).
- Threads are useful for distributed memory service; i.e. a thread can listen to incoming invalidations while another thread runs the computation.
- Threads are also useful in segregating functionality—even when threads are not really necessary. For example, in Chime, after a page fault, the faulting thread suspends itself, while a service thread acquires the page, installs it and restarts the faulting thread.

Windows NT support a variety of thread synchronization and thread control facilities, including the ability of one thread to stop another thread and load or store the other thread's context.

7.4 Process and Stack Migration

Task migration has been used in Calypso to implement pre-emptive scheduling, a topic outside the scope of this paper. Similar mechanisms have been used in Chime for setting the correct scope of tasks.

Suppose a process is executing, and we need to freeze it and restart it on another machine. Under Unix, we would send it a signal and let the signal handler handle the migration.

[4] Allocation of address space is different from allocating memory. A range of addresses can be allocated, without allocating the backing store.

Under Windows NT we use two threads for this purpose [MD97]. One thread listens to incoming messages while the other thread executes. We send a message to the listening thread. This thread suspends the executing thread and extracts its context and then ships the context over to another listening thread on the remote machine. The remote thread sets up the context of a new thread and starts it.

Similar mechanisms are used in Chime. In CC++ it is necessary for two parallel computations to share variable declared in the context of the function that started the parallel computations. This requires a "*distributed cactus stack*". The overview of the implementation steps for this case has been described in section ??. The actual mechanisms used are events to block and restart threads, and the API calls `GetThreadContext` and `SetThreadContext`. We are very happy to see that a thread context saved on one machine, *can be restored on another machine and the thread executes correctly.*

However, in general thread migration is very tricky in Windows NT due to the structure of the system. If a thread is in a DLL and its context is saved, can this context be restored on anther machine? We think not! Some tests reveal that this sometimes works and sometimes does not. However, how to find a "safe" place to save the context of an executing thread (without modifying user-written code) is still an open problem for us.

7.5 Networking

Networking with TCP-IP is almost identical under Unix and Windows NT, via the use *Microsoft Windows Sockets* that provide a similar interface and functionality as that of Berkeley sockets. The Calypso communication module was compatible with the Microsoft Windows Socket interface with the only exception being the asynchronous mode of communication. The asynchronous mode (`FASYNC`), on UNIX, enables the `SIGIO` signal to be sent when I/O is possible. Windows socket implementation has tied the asynchronous mode with the event-driven windows programming. When a socket is in asynchronous mode and I/O is possible, instead of raising an exception, a message is sent to the window specified in the `WSAAsyncSelect()` function call. In effect, the mechanism is synchronous, as the thread processing messages has to read the message synchronously inside an *event-loop*. Moreover, a *console application* can not use sockets in this mode.

7.6 Remote Execution

Creating distributed computations under Unix is simple due to the remote shell (`rsh`) feature. A process can easily spawn more processes on remote machines. Such a feature is not available under Windows NT, making distributed computations use some form of kludge. The preferred way of distributing the computation is to use RPC. While the RPC model is fine for client-server computations, its does not work well for "push" computations such as parallel processing.

We have used a temporary kludge, where a daemon process is started on the machine that will host the tasks of the parallel computation. This daemon listens to commands on a port and starts a process when instructed to do so. A better solution is to have a Windows NT "service" which starts up at boot time and then starts processes using the user-id of the remote user. We have experimented with such a service but have not tested it thoroughly yet.

7.7 Graphical Interfaces

As stated earlier, we used the console application feature to write most of our applications. That is, the application works in a "command window", which looks like the DOS shell, and in some ways similar to an `xterm`. However, all applications under Windows NT are expected to be integrated with the Windows GUI. For our GUI based interface, we cheated and used a Unix-like solution. A separate process runs the GUI and communicates with the controlling process (or manager) via messages, displaying the status and accepting commands. This works well, but is not a politically correct approach under Windows NT.

We are working on developing an event-driven framework that interfaces with MFC and other features in Windows NT to provide a better solution. We have not yet gained enough experience to make this part work (and we are systems programmers and not GUI experts).

8. Performance

The performance we obtained, both under Calypso and Chime were good, and comparable to performance obtained under Unix. For the speedup tests we took a RayTrace program and compiled a sequential version with all compiler optimization turned on. Then we took a Calypso (parallel) version and compiled it under the same optimizations. Then we ran the program on Pentium 90MHz machines and noted the wall clock times. The results are shown in Figure 1.

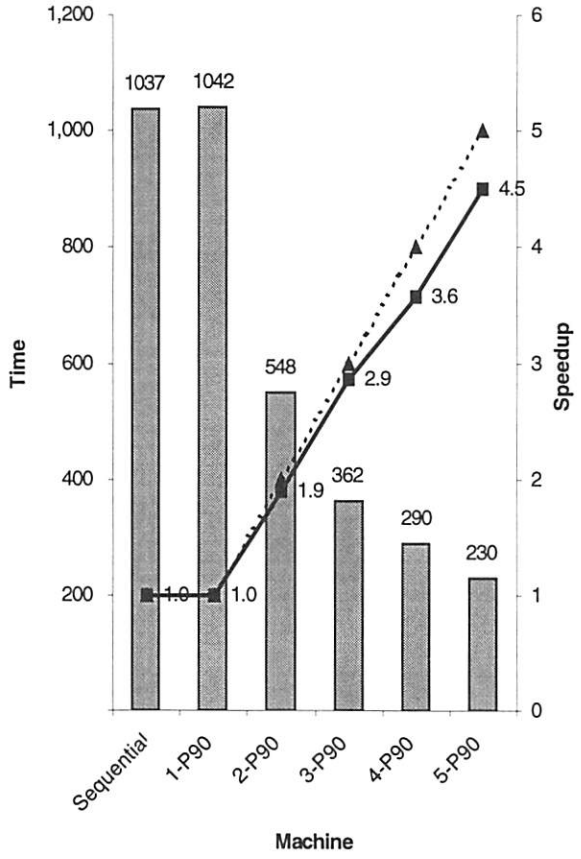


Figure 1: Speedup of Calypso

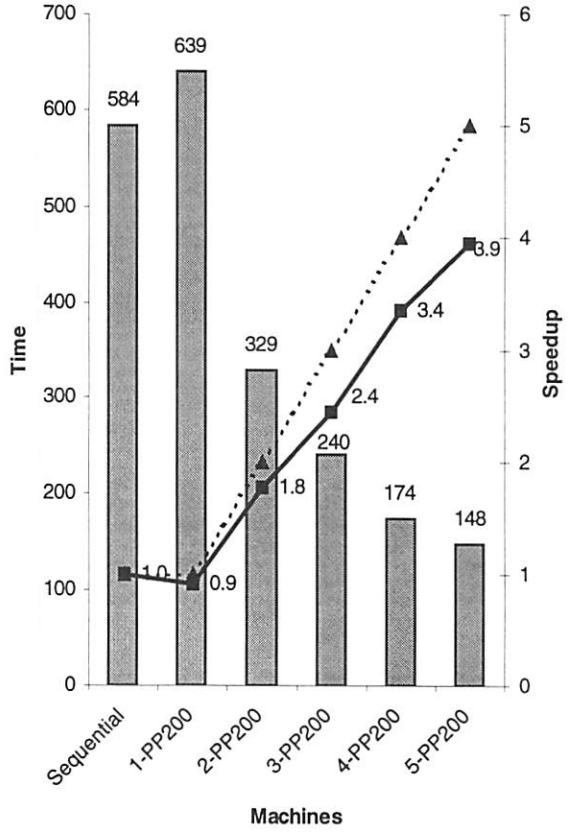


Figure 2: Speedup of Chime

The Calypso program took 1042 seconds to execute on one machine as opposed to 1037 seconds for the sequential program. This shows the low overhead of our mechanisms. We obtained a speedup of 4.5 on 5 machines in spite of providing load balancing and fault tolerance, showing these can be incorporated without additional overhead. This compares very well with Calypso on Unix, which produced the same speedup.

The Chime system is much more complex, and hence produces slightly lower performance. Figure 2 shows the result of running a similar (but larger) RayTrace program under Pentium-Pro 200MHz systems

We performed many other tests, including tests for load balancing (mixture of slow and fast machines) and fault tolerance (transient machines.) Many of these results can be found in [MSD97, SD97] and on the Web site.

A particular test of Chime is interesting. We ran a trivial program under Chime that takes a 1024 element array and initializes each element in parallel. To make the test rigorous, we did thread creation recursively, that is at the top level two tasks are created, which in turn creates two tasks each, until 1024 leaf tasks are created. The total number of tasks created by this program is 3068.

The resulting execution times on varying numbers of machines are shown in Figure 3.

This test shows that the task creation overhead varied from a high of 133 msec and saturates at about a low of 75 msec. The decrease in the time as machines are added is due to some parallelization of the overhead, while the asymptotic value is when the central manager saturates.

9. Further Down the NT Road

Windows NT has much more to offer than the features we have used. Many of the features are beneficial to application programmers and not quite to middleware service providers like us. But many of the more basic features are quite useful and interesting, although the learning curve is steep.

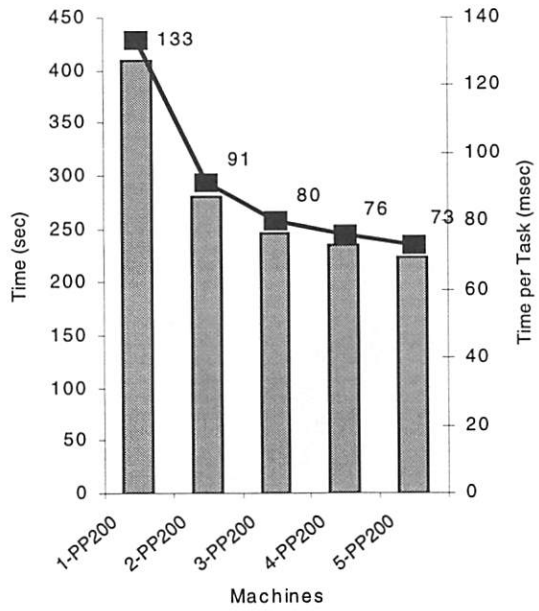


Figure 3: Task Spawning Overhead

Some of the features we would like to explore are:

- Microsoft Foundation Classes
- File system enhancements
- Device Drivers
- Services

The Microsoft Foundation Classes (or MFC) is indeed a powerful, somewhat intuitive (and somewhat confusing) array of prepackaged classes that make programming windows easier.

All our applications were console applications. A windows application has a built in message pump and event handlers. Since the Calypso/Chime systems are essentially libraries, use of a MFC based application is not precluded—however there is need for some modifications. Currently the library provides its own "main" program that set up the memory handling and the memory protection. This is not permissible under MFC programming.

We are looking into how to ensure MFC compatibility. This is not quite straightforward as our managers and workers all have to be automatically generatable from a user program. And the message pumps and the user interfaces need to be created as default, if the user does not specify them, or allow the user to build his/her own interface. This project will be investigated in the future.

Similarly, use of asynchronous I/O, various types of files (mirrored etc.) has advantages that we may be able to exploit. Loadable device drivers will allow us to use custom, lightweight, networking protocols, thus reducing parallel computation overhead.

Some of the features that we did not find a need for include COM, MAPI/TAPI, OLE and other eclectic and fancy support for high end application development.

10. In Retrospect

Windows NT has some very strong points. These include:

- Threads
- Structured Exception handling
- Good memory management
- Excellent program development environment.
- Huge library of online documentation.

And some shortcomings:

- No signals
- No remote execution facility. Main reasons include the manner in which the Windows GUI is structured and the lack of any ASCII support for applications (all applications are GUI applications). This leads to the lack of a `ptty` interface and hence the lack of network logins. While this shortcoming is expected to be fixed in the future with network-aware GUI interfaces.
- Confusing terminology that steepens the learning curve.

Overall, we were happy and impressed. The learning curve was sometimes steep and mostly curvy. Terminology differences were exasperating.[5]

We had to change a lot of programming strategies to suit the Windows NT way of doing things. But, in retrospect the changes we made were for the better.

- Using threads instead of signals for asynchronous event handling is better programming practice.
- Structured Event Handling - very weird when we first saw it, is a nice method of handling exceptions.

The integrated compiler/debugger/makefile system provided by Visual C++ was a wonderful tool to use. The debugging support for multi-threaded programs is fascinating, and without it, we would not ever had process migration to work.

[5] We received a CD-ROM entitled "Microsoft Developer's Library". We thought it contained some library routines that some developers would like to re-use. "*Not for us*", we thought. Turned out, it was chock full of books and articles—some really very useful. A real library!

In addition to the program development environment, of course NT opens up the world of PC computing applications. Office productivity tools, web development tools, personal productivity tools, shareware and freeware are readily available and of great quality. This was an added bonus.

So the final word is that all of the people working on the project unanimously state that is was a nice refreshing move from Unix to Windows NT. NT is a lot nicer system than what we had heard when we fist entered its maze of twisty little passages

11. Acknowledgements

The author wishes to acknowledge the members of the team who worked hard on making everything work on Windows NT. They include Donald McClaughlin, Shantanu Sardesai, Rahul Thombre, Alan Skousen, Siva Vaddepuri and Mahesh Gundelly.

12. Sponsor Acknowledgement/Disclaimer

This research is partially sponsored by the following:

- Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320.
- The National Science Foundation under grant number CCR-9505519.
- Intel Corporation.
- Microsoft Corporation (software donations).

The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

13. Availability

The Calypso NT system is available, for free, complete with documentation, user manual, sample programs, instructions, user interface and remote execution daemon at http://www.eas.asu.edu/~calypso. The Chime system will be available at the same site at a later date.

14. References

[ACD+95] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer,* December 1995.

[AS91] Brian Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. *Workshop on Research Directions in High-Level Parallel Programming Languages*, Mont Saint-Michel, France June 1991.

[BCZ90] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 2nd Annual Symp. on Principles and Practice of Parallel Programming*, Seattle, WA (USA), 1990. ACM SIGPLAN.

[BDK95] A. Baratloo, P. Dasgupta, and Z. M. Kedem. A Novel Software System for Fault Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, 1995.*

[BJ87] K. P. Birman, and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions of Computer Systems*, Vol. 5, no. 1, pp. 47-76.

[BKKW96] A. Baratloo, M. Karaul, Z. Kedem and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Intl. Conf. on Parallel and Distributed Computing Systems*, 1996.

[BL97] R. D. Blumofe, and P. A. Lisiecki Adaptive and Reliable Parallel Computing on Networks of Workstations, *USENIX 1997 Annual Technical Symposium*, 1997

[BS93] D. Bakken and R. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *Technical Report TR93-18*, The University of Arizona, 1993.

[CG89] N. Carriero and D.Gelernter. Linda in Context. *Comm. of ACM*, 32, 1989.

[CK92] K. M. Chandy and C. Kesselman, *CC++: A Declarative Concurrent, Object Oriented Programming Notation*, Technical Report, CS-92-01, California Institute of Technology, 1992.

[DKR95] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems, 1995.*

[DLA+90] P. Dasgupta, R. J. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *IEEE Computer*, 1990.

[GBD+94] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, 1994.

[GJK93] David Gelernter, Marc Jourdenais, and David Kaminsky. Piranha Scheduling: Strategies and Their Implementation. *Technical Report 983*, Yale University Department of Computer Science, Sept. 1993.

[GLS94] W. Gropp, E. Lusk, A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. *MIT Press, 1994*, ISBN 0-262-57104-8.

[HPF93] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0, May 1993.

Also in Scientific Programming, Vol. 2, No. 1 and 2, Spring and Summer 1993; also Tech Report CRPC-TR92225, Rice University.

[JA91] R. Jagannathan and E. A. Ashcroft. Fault Tolerance in Parallel Implementations of Functional Languages, In *The Twenty First International Symposium on Fault-Tolerant Computing*. 1991.

[JF92] R. Jagannathan and A. A. Faustini. GLU: A Hybrid Language for Parallel Applications Programming. Technical Report SRI-CSL-92-13, SRI International. 1992.

[K96] Dilip R. Khandekar. Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems. Master's Thesis. University of Utah. March 1996.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, pages 94-101, August 1988.

[LFS93] J. Leon, A. Fisher, and P. Steenkiste. Fail-safe PVM: A Portable Package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, CMU, 1993.

[MD97] D. Mclaughlin and P. Dasgupta, Distributed Context Switching: A Technique to Speed up Parallel Computations. Available via www.eas.asu.edu/~calypso

[MMB+94] A. Malony, B. Mohr, P. Beckman, S. Yang, F. Bodin. Performance Analysis of pC++: A Portable Data-parallel Programming System Scalable Parallel Computers. In *Proceedings of the Eighth International Parallel Processing Symposium*, pp. 75-85, 1994.

[MSD97] D. Mclaughlin, S. Sardesai, and P. Dasgupta. Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks, *Technical Report, TR-97-001, Department of Computer Science and Engineering, Arizona State University, 1997.* Also available via www.eas.asu.edu/~calypso

[NAB+95] J. Nagib, C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-user Environment. *Technical Report CMU-CS-95-137,* Carnegie Mellon University Department of Computer Science, 1995.

[Pat+94] D. Patterson et.al. A Case for Networks of Workstations: NOW, *IEEE Micro*, April 1996.

[PWC+81] G. Popek and B. Walker and J. Chow and D. Edwards and C. Kline and G. Rudisin and G. Thiel, LOCUS: A Network Transparent, High Reliability Distributed System, *Operating Systems Review*, 15(5), pp. 169-177, Dec 1981.

[R95] Jeffery Richter, Advanced Windows: The Developers Guide to the Win32 API for Widows NT 3.5 and Windows 95 , Microsoft Press, Redmond, WA, 1995.

[S90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.

[SB97] E. Speight and J. K. Bennet, Brazos: A Third Generation DSM System, USENIX Windows NT Workshop, 1997.

[SD97] S. Sardesai and P. Dasgupta, Chime: A Versatile Distributed Parallel Processing Environmen, *Technical Report, TR-97-002, Department of Computer Science and Engineering, Arizona State University, 1997.* Also available via www.eas.asu.edu/~calypso

OPENNT™: UNIX® Application Portability to Windows NT™ via an Alternative Environment Subsystem

Stephen R. Walli
Softway Systems, Inc.
185 Berry Street, Suite 5514,
San Francisco, CA 94107
stephe@opennt.com

1. The Problem

Walli's First Law of Applications Portability: Every useful application outlives the platform on which it was developed and deployed.

Application source code portability is one of the cornerstones of most open systems definitions. The intention is that if an application is written to a particular model of source-code portability, it can port relatively easily to any platform that supports the portability model. This model is often based on source code portability standards such as the ISO/IEEE family of POSIX standards [1,2] and ISO/ANSI C[3], and specifications that include these standards such as the Open Group's Single UNIX Specification[4].

The strength of this model is that the investment in the application's development is not lost in re-deployment to new architectures. Re-writing the application to new platforms every five years or so is not an option. There are already too many new applications to be written in the queue without adding the burden of costly re-writes every few years to address newer faster hardware platforms.

Currently businesses are faced with the decision of moving to Microsoft's Windows NT operating system. Windows NT provides a robust environment to solve many business application problems through a host of software development tools, as well as a platform to run the many Win32-based applications that businesses use today. It does this in a price competitive manner with respect to the hardware platforms on which it runs. The problem becomes protecting the huge investment in applications development over the past decade or more in UNIX applications. How does one leverage and protect the existing application base while moving to Windows NT?

2. Alternatives

There are several ways to move existing applications to Windows NT. These range from the expense of a complete re-write of the application to some form of application port. We will briefly look at the pros and cons of the following:

- a complete re-write of the application to the Win32 environment subsystem
- the UNIX emulation library approach to porting the application
- the common library strategy for porting applications
- the Microsoft POSIX subsystem
- the OPENNT subsystem

2.1 A Brief Introduction to the Windows NT Architecture

Before discussing the alternatives, one must understand a little of the overall Windows NT architecture[5]. The Windows NT operating system was structured such that there is a small kernel (not unlike the design of Mach) that is written such that much of the OS is architecture-neutral with the architecture-specific functionality being provided through the Hardware Abstraction Layer (HAL).

Above the kernel sits subsystems that provide functionality to applications. These subsystems are loosely grouped into functional subsystems (e.g. the Security subsystem, the I/O subsystem) that provide basic functionality, and environment subsystems (e.g. Win32 subsystem, the Microsoft POSIX subsystem, the OS/2 subsystem) that present a programmatic and user interface to developers and users. The primary environment subsystem supported by Microsoft, and regarded as the prescribed way to develop applications on Windows NT is the Win32 subsystem. The Win32 subsystem is also responsible for managing the desktop.

Applications programs essentially run as clients of their respective environment subsystem using a variety of techniques to communicate with the subsystem when requesting operating system services. A fast local procedure call (LPC) and shared memory are two of the more common methods of communication. When an application is started, the executable image is inspected to determine which subsystem will provide the application's services.

2.2 Caveat Lector

Prior to discussing the alternatives and the OPENNT subsystem, it is important to note what this paper is not about. Win32 and traditional UNIX applications use different interfaces for accomplishing the same functions (opening files, creating new processes, etc.). These two operating environments also use different philosophies with respect to how applications are structured when looking at such items as the handling of standard I/O streams, the use of file descriptors, consistent return codes, and process creation relationships and control.

This paper does not get into religious or philosophical debate about which OS interface or application architecture may or may not be "better" than the other. Neither does it descend into "marketing" messages about the rationale for providing two different programming and run-time environments on Windows NT. This paper is about solving a very real application programming problem using a unique strategy based on the fundamental design of the Windows NT OS.

2.3 Application Rewrite to Win32

There is a family of simple application programs that are written to ANSI C and its library specifications and will simply re-compile on the Windows NT operating system in the Win32 subsystem space.

Most applications present a greater challenge as they use some form of system resources. Such applications require some amount of rewrite to address the lack of interfaces that directly map the traditional UNIX application interface to which they were written. Some of the re-write may be very straightforward if the use of system services was restricted to simple file operations (*open( ), close( ), read( ), write( )*). If the application depends upon complex signal mechanisms, parent-child relationships (*fork( ), exec( )*), or absolute file semantics (hard links, or the ability to distinguish files by case alone), then the rewrite strategy becomes more expensive as parts of the application must now be re-designed, then re-written.

When one considers a limited number of application programs, this may be an option. If however, one is reviewing a large number of applications, the cost of rewriting can become excessive.

2.4 Win32-based UNIX Emulation Libraries

There are a number of Win32-based libraries which emulate the behavior of UNIX system calls and libraries. The most common include:

- UWIN from AT&T Laboratories
- NuTCracker from Datafocus
- Portage from Consensys

Please see Korn [6] for a comparison of these three packages, and for a complete discussion of the UWIN package.

A set of applications may compile cleanly in this environment, but may not run correctly because system calls don't demonstrate the expected semantics. For example, the semantics for *fork()* and *exec( )* cannot be exactly duplicated with respect to the process attributes inherited by the child process. NuTCracker and Portage leave a process "ghost" around, creating a new process, when an *exec()* call is issued. To obviate the need for exact semantics the UWIN project created a *spawn( )* call that replaces the previous two interfaces with a single interface.

File system semantics are another challenge. The Windows NT file system (NTFS) provides almost complete UNIX file system semantics. Group ownership of files, hard links, and filename case sensitivity is all available from NTFS, but are not available through the Win32 subsystem.

All these emulation systems allow for (indeed may require) the use of Win32 interfaces in the application source code. This is not a feature. In a less experienced development operation it can destroy the application's portability, locking it to the hybrid environment. At best it increases the maintenance cost of the source with an additional level of conditional compilation, and the work may not easily integrate into the existing style of UNIX code.

2.5 *Common Porting Libraries*

Depending upon the application space there may be a set of libraries that have already been created that are used as the portability layer.
The only requirement here is that the library exists on Windows NT. The library layer then becomes the porting exercise. This is a special case of the normal situation where the application source code needs to be ported or re-written. It is an application structuring issue.

2.6 *The Microsoft POSIX Subsystem*

Windows NT supports multiple environment subsystems by design. One of the original subsystems was the POSIX subsystem. The Microsoft POSIX subsystem is an exact implementation of the ISO/IEEE POSIX.1 standard, which includes the ANSI C library by reference.

Microsoft down-played the POSIX subsystem, essentially pointing out that it was delivered for conformance to NIST FIPS Pub 151-2 [7] requirements for government agency procurement. The development environment was not very supportive, and it was poorly documented. As an exact implementation of POSIX.1, the Microsoft POSIX subsystem was somewhat limited in functionality.

The interesting thing to note about early experiments with the Microsoft POSIX subsystem is that all the "big" things (process semantics, signals, and the file system) behave as expected – it is the little things that are surprising. The *ttyname( )* description in POSIX.1 states that the function returns a pointer to a string containing the pathname of the terminal, but returns a NULL pointer if the file descriptor is invalid or if the pathname cannot be determined. There are no error conditions that are detected[1]. A function that always returns a NULL pointer is a sufficient implementation to fulfill the letter of the standard without being particularly useful to an application.

That said, the Microsoft POSIX subsystem passed the breadth of the NIST FIPS 151-2 certification process. In fact, the original OPENNT Commands & Utilities [8] shipped in March 1996 were built on top of a subsystem that was for the most part the original Microsoft POSIX subsystem. That achievement demonstrated that the environment subsystem architecture of Windows NT was flexible and powerful.

2.7 *The OPENNT Subsystem*

If the Microsoft POSIX subsystem could be built then why not a complete "UNIX" environment subsystem? There is nothing in the Open Group Single UNIX Specification and its attendant UNIX 95 [9] brand that can not be implemented on top of the Windows NT kernel. As you will see, this also applies to the graphic environment of the MIT X11 project.

The goals when developing OPENNT were:

- To provide a complete porting and runtime environment to migrate applications source code developed on traditional UNIX systems directly to Windows NT. This meant going beyond the standards and specifications (e.g. providing X11R5) as well as providing more than one way to access functionality (e.g. determining the next available master pseudo-terminal).
- To provide true semantics for the system interfaces such that application source code would not need to change to account for "not quite UNIX" semantics.
- To ensure any changes made to the application source code should make it more portable (i.e. follow the standards) rather than less portable (i.e. using Windows NT specific constructs.)
- To ensure performance was not effected by an appreciable amount.
- To ensure that the Windows NT operating system's integrity was not compromised in anyway (e.g. security).
- To integrate the OPENNT subsystem cleanly into the Windows NT world such that it was not "isolated" with respect to such things as data access and application or system management.

After initial investigations into Windows95, it was decided to not pursue a solution in this space. There was no way to provide true POSIX/UNIX semantics on Windows95.

The overall goal was to leverage the Windows NT environment subsystem architecture and design to its logical conclusion, taking it from the general purpose application environment that exists solely in the Win32 subsystem space, and using it to provide an application platform for both Win32 and UNIX-based applications.

3. A Short History of OPENNT

In September 1995, Softway Systems, Inc. entered into a long-term agreement with Microsoft to extend the Microsoft POSIX

subsystem into a complete traditional UNIX subsystem, capable of branding to the Open Group's UNIX 95 profile brand. Work began immediately to wrap the POSIX subsystem in a shell and utilities environment (delivered March 1996) that conformed to the POSIX.2 Shell and Utilities Execution Environment. An X11R6 server was added (July 1996), along with a telnet service (August 1996). A software development kit (SDK) was added such that developers could begin porting their own applications (September 1996). At that time, the SDK essentially supported the POSIX.1 interfaces, ISO C library, and approximately 60 historical Berkeley library routines from the 4.4BSD-Lite distribution that were required while developing the first POSIX.2 implementation. It provided the additional tools required to develop applications (the RCS version control suite, *ar*, *cc/c89*) and wrapped the Microsoft Visual C/C++ command-line compiler.

Extending the OPENNT subsystem to map more of the kernel interface and adding libraries and utilities has been ongoing. With the release of OPENNT 2.0 (May 1997), there is now support for:

- POSIX.1, including a nearly complete general terminal interface
- ISO C standard library
- Berkeley sockets
- System V Interprocess Communications facilities (shared memory, semaphores, message queues).
- Memory mapped files
- System V and Berkeley signal interfaces layered onto the POSIX.1 signal semantics.
- traditional curses (ncurses — also supports colour)
- X11R5 clients libraries (Xlib, Xt, Xaw, etc.) and almost all the X11R5 clients
- pseudo-terminals
- OSF/Motif 1.2.4
- cron service
- full job control in the shells (ksh, csh)
- tape device support
- Perl 5
- the public domain KornShell with full job control, the Tenex csh and 200+ utilities

4. The OPENNT Architecture

The overall architecture of OPENNT consists of the environment subsystem, its mapping to the Windows NT file system (NTFS), its relationship to functional subsystems (e.g. security), and its compiler environment. Issues of integration with the rest of Windows NT and the Win32 subsystem are discussed in the following section.

4.1 The OPENNT Subsystem

The OPENNT subsystem consists of three parts:

- The subsystem itself (PSXSS.EXE) that functionally re-maps the Windows NT kernel and manages such items as process relationships and signal delivery.
- The terminal session manager (POSIX.EXE) that manages the desktop console window for each OPENNT-based session leader.
- The dynamic link library (PSXDLL.DLL) that handles certain system service requests directly, as well as the communication between the subsystem and OPENNT processes.

When an OPENNT application is started for the first time, the Win32-based Program Manager determines that the application belongs to a subsystem other than the Win32 subsystem and a number of events happen:

- If the OPENNT subsystem is not already running, it is started. The subsystem can be configured to start-up at system boot time, but by default is started for the first OPENNT application.
- A terminal session manager is started to manage console window output for the application. A terminal session manager runs for each session. It is the tty device for an OPENNT application.

At that point, application processes run, communicating with the OPENNT subsystem in the same manner as a Win32 application runs and communicates with the Win32 subsystem. Fast Local Procedure Call (LPC) and shared memory are the two primary mechanisms used for communications between the OPENNT subsystem and its processes.

4.2 The Windows NT File System

NTFS was developed for Windows NT to provide a fully functional file system and address the shortcomings of the DOS FAT file system. It provides:

- Recoverability and redundancy of critical disk structures
- Enhanced security consistent with the Windows NT security model
- Data redundancy capabilities to support disk mirroring and striping
- Support for large volumes
- Unicode-based filenames

- Bad cluster remapping
- POSIX file system semantics, including case sensitivity for filenames, hard links, the file change time stamp, and group ownership

File access is mapped from the process to data on the disk via a layered driver model. The NTFS driver is simply another driver layer under the control of the Windows NT executive I/O manager. The NTFS driver can further layer on various fault tolerant drivers, and ultimately layers onto the actual disk driver.

All system service calls for file access come through the I/O manager, regardless of the environment subsystem of origin, i.e. both the Win32 and OPENNT subsystems share a common view of the NTFS. There is no container file system in which POSIX/UNIX file system semantics are emulated. Files are treated as objects within the Windows NT executive, and are managed by the Object manager with respect to object (file) sharing and protection. The Object manager interfaces with the Security Reference Monitor when checking file access permissions.

While the OPENNT subsystem shares a common view of the files within the NTFS, additional functionality is supported. The OPENNT subsystem is able to create hard-links in the file system, something not available via the Win32 subsystem. Win32 subsystem applications (such as the Explorer or File Manager) see two separate files. Case sensitivity in the filename is supported through the OPENNT subsystem, so both *makefile* and *Makefile* can exist, and removing one will not accidentally remove the other.

Advanced features of NTFS are also available to applications running from the OPENNT subsystem. The audit capabilities to track file access success/failure of open, close, read, and write operations are features of NTFS, and available to OPENNT applications. The ability to use additional permission controls with access control lists is also available. These are all file system capabilities that managed by applications that are outside the typical ported application. Windows NT provides the tools to manipulate the audit and advanced security features of NTFS.

4.3 Security, Privileges, and Permissions

Windows NT has obtained its U.S. Department of Defense Orange Book C2 security certification, supporting all required discretionary access controls. One of the goals of OPENNT was to ensure it worked cleanly with the security model on Windows NT, and did not compromise any of its capabilities.

In general, objects (e.g. files) are protected by access control lists (ACL) that are made up of access control entries (ACE). When a user logs onto Windows NT, authentication is provided via username and password and confirmed through the security subsystem. An access token is associated with the successfully logged on user's process and this is used in all actions with objects to determine what access is appropriate given the objects' ACLs.

Several issues arise with respect to OPENNT. The user and group name space is shared. There is no concept of separate user and group databases. Groups can also own objects. This means that a file created on the Win32 side of the house may be owned by a group, and a long listing (*ls*) from the OPENNT side of the house will show a group name as both the owner and the group of the file. This is a little disconcerting the first time it is witnessed. Files created from the OPENNT side of the house assign ownership appropriately and consistently with the POSIX standards.

There are no actual permission bits for a file or directory, but rather ACLs are used to map the permission world of UNIX/POSIX. There is an ACE entry for the file owner, file group, and a group named "Everyone" that align with the appropriate permission fields traditionally associated with file permissions in the UNIX world. The ACL can be displayed via the File Manager. Again, files created from the Win32 side of the house will receive an ACL consistent with the Win32 subsystem rules. A long listing from the OPENNT world must map the ACL to permission bits as best it can. Additional levels of security are possible by adding additional ACEs.

There are no */etc/passwd* or */etc/groups* files. All authentication information is kept in the database accessed by the security subsystem. This is one of the two areas that source code changes specific to OPENNT are required for an application that authenticates users. (The other is handling rooted pathnames that expect /usr and /bin to exist in the file hierarchy.) As there is no password database against which to

authenticate a user, OPENNT provides a simple functional interface to handle user authentication, as well as a set of *exec( )* functions to execute an application as a particular user. The OPENNT login program executed by the OPENNT telnetd makes use of these functions to setup the shell as a new user.

Windows NT does support an Administrator user with enhanced privileges, as well as an administrators group. There is, however, no root user with all privileges. This has not caused any insurmountable problems in any of the application porting experiences we have had to date, nor have any of the customer base or product beta-testers complained about this lack of a root user id.

4.4 The Compiler Environment

The OPENNT development environment consists of a set of headers, libraries, and shell script wrappers around the command-line version of the Microsoft Visual C/C++ compiler (CL.EXE) and linker (LINK.EXE).

The compiler builds object modules that can be linked regardless of the subsystem with which the application will run. The Microsoft linker knows how to stamp the executable appropriately for the "POSIX" subsystem, such that it is correctly passed off to the subsystem when the Win32 subsystem determines it is a "POSIX" binary at application start-up time.

Programs that run as clients of one environment subsystem cannot make calls to interfaces supported by another environment subsystem, so separate libraries are provided to ensure no dependencies to the Win32 world are referenced within the libraries.

As of publication time, Softway Systems developers are completing a port of *gcc* into the OPENNT environment. Both Intel and DEC Alpha platforms are being supported by the *gcc* tool set. The compiler suite will be made available via the Tool Warehouse on the Softway Systems, Inc. web site:
http://www.OpenNT.com

4.5 Performance Issues

The OPENNT subsystem is a peer environment subsystem to the Win32 subsystem. OPENNT processes communicate with the OPENNT subsystem using the same mechanisms as Win32 subsystem processes use to communicate with the Win32 subsystem.

Comprehensive benchmarks have not been run, but early informal work has been done using the iozone [9] benchmark, netperf [10] benchmark, and some informal programs to compare CPU bound program throughput. The iozone tests and CPU tests were run on a traditional UNIX system as well as both the Win32 and OPENNT subsystems running on identical hardware. The netperf benchmarks were run between comparable Windows NT machines, and Windows NT and a similar Intel machine running a traditional Berkeley-based system. Informal trials indicate:

- CPU bound applications perform the same between the Win32 and OPENNT subsystems.
- CPU bound applications performed better on OPENNT than a mainstream traditional UNIX system.
- OPENNT and a traditional UNIX platform showed comparable disk performance trends with iozone.
- For small block *read( )* and *write( )* operations, Win32 outperforms OPENNT. For large block *read( )* and *write( )* operations, OPENNT outperforms Win32.
- Socket throughput between Windows NT and a traditional Berkeley system on a local network is virtually the same regardless of whether the performance testing programs are running with the OPENNT or Win32 subsystems.

Performance tuning is an ongoing task and there are a number of projects underway to increase the overall performance of the OPENNT subsystem.

5. Integration with the Win32 World

Walli's Second Law of Applications Portability: Useful applications seldom live in a vacuum.

There are a number of stresses on applications that exist on multiple platforms or have a history of being migrated to new platforms.

- Most operating systems provide functionality beyond that defined in the POSIX family of standards and the Open Group specifications. This additional functionality may be as straight forward as providing the X11 GUI, or as complex as MVS, VMS, and Windows NT, where another entire operating environment is present.

- Applications often contain platform specific source code. This can be to leverage platform specific functionality, or to handle

functionality provided differently on different platforms.

- Once deployed the need to share data between applications becomes a necessity, and new applications are created in the space between existing applications.

- The use of non-standard or platform specific tools and functions is sometimes necessary to solve the business problem, indeed this may drive the actual platform purchase.

Determining the balance between protecting the application investment and solving the business problem with a platform's unique attributes becomes a challenge.

Windows NT provides a rich environment of tools and functionality, developed on top of the Win32 subsystem. The model for integration between the Win32 and OPENNT worlds happens at a higher level than the application programming interface. Applications source code is ported directly to Windows NT to run with the OPENNT subsystem. This protects the application investment. Integration with the Win32 world can then take place in a number of ways: NTFS, the Desktop, Win32exec, and sockets.

From an end-user's point of view, on a single machine environment they simply have a "desktop" full of applications. They don't care which subsystem the application is communicating with for kernel services anymore than they care in which language the programmer wrote the source code.

5.1 NTFS

The Win32 and OPENNT worlds share a common file system. Files created in one world are seen in the other. All the security and auditing features provided by NTFS and managed through the Win32-based administration tools are available to OPENNT ported applications.

5.2 The Desktop

Windows NT presents either the Windows 3.1 or the Windows 95 desktop. This environment easily supports interaction between the OPENNT and Win32 worlds.

An OPENNT subsystem terminal or tty is a Win32 console. This means large windows and screen buffers, scroll bars and cut-and-paste are all available at the user interface. An OPENNT console window behaves very much like a local *xterm*. Cut-and-paste between Win32-based applications and OPENNT applications is flawless. For example, text from a Microsoft Word document can be easily cut then pasted into a *vi* session or OPENNT shell.

Icons can be set up to launch OPENNT applications, included X11-based applications and shell scripts.

Win32 GUI applications launched from an OPENNT shell present their own window as would be expected, and the application can either be run in the foreground (where the shell will wait for it to complete) or the background.

5.3 Win32exec

An early ability demanded by users was to execute Win32-based applications from the OPENNT world. A Win32exec ability was added that allows Win32 GUI and command-line applications to be executed from within an OPENNT process.

There are a number of challenges to overcome in this space. A Win32 GUI application is relatively straight forward, in that there is no I/O to be managed between the subsystems. Once control of the process has been passed to the Win32 subsystem, the OPENNT shell can wait or continue as desired. If a Win32 command-line user interface (CUI) application is run, and its output is to be seen in the shell window, as if from the standard output of a "normal" child process rather than one running as a client of another subsystem, a certain amount of handshaking and re-direction needs to be performed. Standard streams need to be mapped to Win32 handles appropriately, such that I/O can be redirected to other processes (Win32 or OPENNT) in a pipeline, or to files.

The ability to execute Win32 applications allows a number of facilities to be instantly accomplished in a manner most consistent with Windows NT, but with a "UNIX" interface.

For example, the *lp* utility becomes a simple shell script wrapper around the Win32 *PRINT.EXE* command. All the functions available on the network printing system are instantly available in a manner most appropriate to the architecture, while providing an interface most appropriate to a traditional UNIX environment. Likewise, *useradd* and *userdel*, traditional UNIX commands to manage users,

become simple scripts wrapped around *NET.EXE*.

5.4 OPENNT Sockets and Winsock

Applications can communicate in client/server fashion using sockets and the TCP/IP protocol. Client/server applications can be built to use either the Win32 GUI for client code, written on Winsock in the Win32 subsystem (run on Windows NT and Windows 95), or maintain the traditional UNIX client code with its X11 or simple character interface written using sockets. Server code is maintained as traditional UNIX code running on the OPENNT subsystem. The long term flexibility of the solution is completely in the developer's hands.

There are a number of examples of the strength and flexibility of this solution in our own product space.

- Early on we ported the Apache web server directly into the OPENNT environment. The Apache server source code plus all the existing Perl, CGI, HTML scripting from the UNIX environment comes directly into the Windows NT world on the OPENNT subsystem. The Microsoft Internet Explorer or Netscape browser are Win32 "clients" that can then be used to connect to the web server.

- The OPENNT X11R6 server is a Win32 application. All the OPENNT X11R5 clients are OPENNT subsystem applications.

- The telnet daemon shipped with OPENNT 2.0 is a direct port of a traditional UNIX telnetd running on the OPENNT subsystem. Any telnet client, local or remote, Win32-based or UNIX-based, can connect to it. (As a further example of mixing and matching in the environment to best express functionality while protecting the application source, the OPENNT subsystem telnetd further runs as a Windows NT service, controlled by either the OPENNT command-line *service* utility or the Win32 GUI service control applet in the Windows NT Control Panel.)

6. Early Porting Experiences

The following sub-sections discuss porting experiences with various packages of software that the Softway Systems developers have ported over time.

6.1 4.4BSD-Lite

Much of the original utility base for the early OPENNT commands and utilities came straight off the 4.4BSD-Lite distribution. The general flow was to copy the source distribution over to a working directory, use a simple template makefile, and begin simple compile-edit cycles until the utility built. The first thing that always needed to be changed was to change <*sys/param.h*> to <*sys/types.h*> and <*limits.h*> as that is the "standard" way to get the types and limits on a POSIX.1 based system. (A skeletal <*sys/param.h*> has since been added to the SDK.) Before sockets and memory mapped files were implemented, any reference to them needed to be stepped around. These functions and macros have since been "turned on" again. Symbolic link code was also avoided. Once linked, testing and conformance work could begin.

6.2 GNU Source

GNU source was used for a number of utilities in the original packages (*RCS*, *diffutils*, *binutils*). The challenge here has always been configure scripts. First, the MSVC command-line compiler always outputs the name of the module being compiled which confuses configure scripts terribly (and there is no way to turn this output off). Configure scripts also make liberal use of argument order to the compiler that while common practice is not "standard". The early c89/cc script complained about this.

Since Win32exec and pipes has been turned on in the subsystem, the c89/cc script has been modified to handle output from the compiler, and changes have also been made to handle the argument order. Configure scripts work much better now. Gmake has also been ported so configure scripts that rely on it have fewer problems.

Configure aside, the source code itself often just builds. Before the configure support was added, the fastest way to port a GNU tool to OPENNT was to toss away the configure script, copy the makefile template and configuration header template, and quickly hand-tune them turning on anything that said POSIX or ANSI C. A fast inspection was often enough to get it right. The source code typically built at that point. For example, the only changes required in the 17000+ lines of *gawk* was to change <*varargs.h*> to <*stdarg.h*> in two places.

6.3 Perl5

The Perl5 distribution was one of the early tools built and made available off the Tool Warehouse section of the OPENNT web page. The distribution is 78,796 lines of code. Running the configure scripts was the only way to determine how to best build Perl5 because of the number of options and permutations available.

The configure script was modified as follows:

- Correct compilation argument order problems (mentioned above).
- Point to the location of the OPENNT header files in *$OPENNT_ROOT/usr/include*.

The configure script produces a set of scripts that further generate the makefiles and configuration-based header files. While the overall configure script could not be run from start to finish due to problems in the environment, it ran well enough to produce the individual sub-scripts which ran correctly, producing a set of headers and makefiles.

The only source changes made stepped around some non-portable use of the user database fields *pw_gecos* and *pw_passwd*. Perl5 passes all the test cases in the test suite shipped with the distribution, with the single exception of the setuid test.

6.4 Apache

Apache, the public domain web server, is 45,726 lines of code. It was ported as an experiment early in the alpha test cycle of OPENNT 2.0.

Initially,

- symbolic links, *setsid( )* and *setpgrp( )* were stepped around.
- Calls to *mktemp( )* were changed to use *getenv( )* of *$TMPDIR* instead of hard-coding path names.
- Uses of *chown( )* were avoided as they violated the *chown( )* functionality mandated by POSIX.1 in association with the standard option *POSIX_CHOWN_RESTRICTED*.
- A *crypt( )* routine (taken from 4.4BSD-Lite) was required.

The following stanza (reformatted for publication) was added to the configuration header for OPENNT.

```
#elif defined(OPENNT)
#define S_ISLNK(m) (0)
#define  bzero(a,b) memset(a,0,b)
#define     USE_FCNTL_SERIALIZED_ACCEPT
#undef   HAS_GMTOFF
#define  NO_KILLPG
#define  NO_SETSID
#define  JMP_BUF sigjmp_buf
#include <sys/time.h>
#define  getwd(d)
     getcwd(d,MAX_STRING_LEN)
#define  SIGURG SIGUSR1
```

This stanza was about the same size as any other system specific stanza, and has since been updated to handle the new functionality provided since the alpha test version of OPENNT 2.0.

6.5 xv

The *xv* utility is the popular X11-based tool for browsing and manipulating graphic images (e.g. GIF and JPEG). It contains a full file browser as part of the utility. The distribution is approximately 83,600 lines of code. A few trivial source changes were required to:

- step around non-portable use of *mknod( )*
- avoid a use of *endpwent( )*
- add a #define to properly use *strerror( )*

xmkmf was used to generate the appropriate Makefile and *xv* was made.

7. Summary

It has become too expensive to continually rewrite applications to move them from system to system, and source code portability is an important tool to protect existing applications investments. Our experience clearly demonstrates that the Windows NT architecture of alternative environment subsystems provides a way to accomplish this. OPENNT provides the facilities of a traditional UNIX system on Windows NT, such that existing applications developed on traditional UNIX systems can be directly brought to Windows NT, rebuilt, and deployed. Using the environment subsystem architecture of Windows NT, a peer environment to the Win32 world exists, and is integrated to that world in a manner most logical to both.

Up-to-date information about OPENNT can be found at: http://www.OpenNT.com

8. References

1. ISO/IEC 9945-1:1990, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C

Language], IEEE Standards, NJ, ISBN 1-55937-061-0
2. ISO/IEC 9945-2:1993, Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities, IEEE Standards, NJ, ISBN 1-55937-255-9
3. ISO/IEC 9899:1990, Programming Languages—C,
4. X/Open CAE Specification:
〈 System Interfaces and Headers, Issue 4, Release 2
〈 Release 2 Commands and Utilities, Issue 4,
〈 System Interface Definitions, Issue 4, Release 2
X/Open Company Ltd., 1994
5. Custer, Helen, Inside Windows NT, 1993, Microsoft Press, Redmond WA (ISBN 1-55615-481-X)
6. Korn, David, *Porting UNIX to Windows NT*, Proceedings of the USENIX 1997 Annual Technical Conference, pp. 43-57, 1997
7. National Institute of Standards and Technology, Federal Information Processing Standards Publication 151-2, *Portable Operating System Interface (POSIX) - System Application Program Interface [C Language]*, 12 May, 1993.
8. OPENNT Commands & Utilities, Release 1.0, Softway Systems Inc., March 1996.
9. Walli, Stephen R., Go Solo: How to Implement and Go Solo with the Single UNIX Specification, Prentice Hall, Englewood Cliffs, NJ, 1995
10. The iozone benchmark available from this site includes source code and documentation for several I/O benchmarks, 9 May, 1997 http://www.cs.umbc.edu/~elm/Ftp/iobenchmarks
11. netperf-1.7.1 Netperf is a benchmark for measuring networking performance. It focuses on bulk data transfer and request/response performance using TCP or UDP and the Berkeley Sockets interface. It is maintained and supported by the IND Networking: 19 March 1997 http://hpux.dsi.unimi.it/hppd/hpux/Networking/Admin/netperf-1.7.1/

9. Trademarks

UWIN – UNIX* for Windows

David G. Korn (dgk@research.att.com)

AT&T Laboratories
Florham Park, N. J. 07932

Abstract

This paper describes an effort of trying to build a UNIX interface layer on top of the Windows NT[1] and Windows 95[2] operating system. The goal was to build an open environment rich enough to be both a good development environment and a suitable execution environment.

The result of this effort is a set of libraries, headers, and utilities that we collectively refer to as UWIN. UWIN contains nearly all the X/Open Release 4[3] headers, interfaces, and commands. An earlier paper on porting to Windows NT[4], describes alternative porting strategies and presents some performance results for UWIN. In this paper we discuss some of the design decisions behind UWIN and some of the results so far as well as some of the remaining challenges.

1. INTRODUCTION

The marketplace has dictated the need for software applications to work on a variety of operating system platforms. Yet, maintaining separate source code versions and development environments creates additional expense and requires more programmer training.

The Software Engineering Research department at AT&T Labs writes and distributes several widely used development tools and reusable libraries that are portable across virtually all UNIX platforms.[5] To enhance reuse of these tools and libraries, we wanted to make them available on systems running Windows NT and/or Windows 95. We did not want to spend the cost needed to support multiple versions of these libraries, and we wanted to minimize the amount of conditionally compiled code.

One way to lower this cost is to use a middleware layer that hides the differences among the operating systems. The problem with this approach is that it forces one to program to a non-standard, and often proprietary, interface. In addition, it often limits one to the least common denominator of features of the different operating systems.

An alternative is to build a middleware layer based on existing standards. This has been the approach followed by IBM with the introduction of OpenEdition[6] for the MVS operating system, URL `http://www.s390.ibm.com/products/oe`. OpenEdition is X/Open compliant so that a large collection of existing software can be transported at little cost.

Windows NT is an operating system developed by Microsoft to fill the needs of the high-end market. It is a layered architecture, designed from the ground up, built around a microkernel that is similar to Mach.[7] One or more *subsystems* can reside on top of the microkernel which gives Windows NT the ability to run different logical operating systems simultaneously. For example, the OS/2 subsystem allows OS/2 applications to run on Windows NT. The most important subsystem that runs on Windows NT is the WIN32 subsystem. The WIN32 subsystem runs all applications that are written to the WIN32 Application Programming Interface (API)[8]. The API for the WIN32 subsystem is also provided with Windows 95, although not all of the functions are implemented. In most instances binaries compiled for Windows NT that use the WIN32 API will also run on Windows 95.

The POSIX subsystem allows applications that are strictly conforming to the IEEE POSIX 1003.1 operating system standard[9] to run on Windows NT. Since the POSIX standard contains most of the

* UNIX is a registered trademark, licensed exclusively through X/Open, Limited.

standard UNIX system call interface, many UNIX utilities are simple to port to any POSIX system. Because most of our tools require only the POSIX interface, we thought that it would be sufficient to port them to the POSIX subsystem of Windows NT. Unfortunately, this is not a viable alternative for most applications. Microsoft has made the POSIX subsystem as useless as possible by making it a closed system. There is no way to access functionality outside of the 1990 POSIX 1003.1 standard from within the POSIX subsystem, either at the library level or at the command level. Thus, it is impossible to invoke the Microsoft C compiler from within the POSIX subsystem. Softway System, Inc., URL `http://www.softway.com`, has an agreement with Microsoft to enhance the POSIX subsystem. Softway claims that they will open up the POSIX subsystem so that it can access WIN32 applications. They have built the OpenNT product with this enhanced subsystem. However, it is still not possible to mix UNIX API calls and WIN32 calls in a single application. A final drawback to this approach is that the POSIX subsystem is not available for Windows 95.

We investigated alternative strategies that would allow us to run programs on both UNIX and Windows NT based systems. After looking at all the alternatives, we decided to write our own library that would make porting to Windows NT and Windows 95 easy. We spent three months putting together the basic framework and getting some tools working. Realizing that the task was larger than a one person project, we contracted a small development team of 2 or 3 to do portions of the library, packaging, and documentation. This paper discusses the implementation of our POSIX library and current status. Version 1.2 of UWIN has been freely available in binary form for non-commercial use on the internet from the web site `http://www.research.att.com/sw/tools/uwin`. Version 1.3, described here, should be available from this site by the time of this conference.

2. GOALS

We wanted our software to work with Windows 3.1, Windows 95, and Windows NT. A summer student wrote a POSIX library for Windows 3.1 and we were able to port a number of our tools. However, the limited capabilities of Windows 3.1 made it a less than desirable platform. We instead focused our goals on Windows NT and Windows 95. We decided to use only the WIN32 API for our library so that the library would work on Windows 95 and so that all WIN32 interfaces would be available to applications.

Initially, our goal was to provide the IEEE POSIX.1 interface with a library. This would be sufficient to run `ksh` and about eighty utilities that we had written. It soon became obvious that this wasn't enough for many applications. Most real programs use facilities that are not part of this standard such as sockets and/or IPC.

We needed to provide a character based terminal interface so that curses based applications such as `vi` could run. After the initial set of utilities was running, we wanted to get several socket based tools working. Several projects at AT&T that became interested in using our libraries, required the System V IPC facilities. The S graphics system[10] and `ksh-93`[11] required runtime dynamic linking. As the project progressed, the need for privileged users, such as `root` on UNIX systems, surfaced. We decided that it was important to have `setuid` and `setgid` capabilities. It soon became clear that we needed full UNIX functionality and we set our goal on X/Open Release 4 conformance.

We needed to have a complete set of UNIX development tools since we didn't want to get into the business of rewriting makefiles or changing build scripts. Most code written at AT&T, including our own, uses `nmake`[12], (no relation to the Microsoft `nmake`), but we also wanted to be able to support other `make` variants. We didn't want to do manual configuration on tools that have automatic configuration scripts.

A second measure of completeness is the degree to which all the system services can be accessed through the UNIX API. Our goal was to be able to perform as many of the Windows NT tasks as possible via traditional UNIX commands. For example, it should be possible to reboot the system when logged in through a telnet session by calling the UNIX `shutdown` program. We should also be able to access the Windows registry database with standard UNIX utilities such as `grep`.

One important goal that we had from the beginning was to not require WIN32 specific changes to the source to get it to compile and execute. The reason for this is that we wanted to be able to compile and execute UNIX programs without having to understand their semantics. In addition we wanted to limit the number of new interfaces functions and environments variables that we had to add to use our library. It is difficult to manage more than one or

two environment variables when installing a new package.

Another goal that we had was to provide a robust set of utilities with minimal overhead. If utilities written to the X/Open API were noticeably slower than the same utilities written to the native WIN32 API, then they were likely to be rewritten making our library unnecessary in the long run.

A final and important goal was interoperatability with the native Windows NT system. Integration with the native system not only meant that we could use headers and libraries from the native system, but that we could pass environment variables and open file descriptors to commands written with the native system. There couldn't be two unrelated sets of user ids and separate passwords. If write permission were disabled from the UNIX system, then there should be no way to write the file using facilities in the native system and vice versa.

We have not as yet achieved all of our goals, but we think that we are close. The rest of the paper will discuss some of the issues we needed to deal with and our solutions.

3. PROBLEMS TO SOLVE

The following problems need to be understood and dealt with in porting applications to Windows NT. These are some of the issues that need to be addressed by POSIX library implementations. Section 6 describes how UWIN solved most of these problems.

3.1 Windows NT File Systems

Windows NT supports three different file systems, called FAT, HPFS, and NTFS. FAT, which stands for File Access Table, is the Windows 95 file system. It is similar to the DOS file system except that it allows long file names. There is no distinction between upper and lower case although the case is preserved. HPFS, which stands for High Performance File System, was designed for OS/2. NTFS, the native NT File System, is similar to the Berkeley file system.[13] It allows long file names (up to 255 characters) and supports both upper and lower case characters. It stores file names as 16 bit Unicode names.

The file system namespace in Win32 is hierarchical as it is in UNIX and DOS. A pathname can be separated by either a / or a \. Like DOS, and unlike UNIX, disk drives are specified as a colon terminated prefix to the path name, so that the pathname `c:\home\dgk` names the file in directory `\home\dgk` on drive `c:`. Many UNIX utilities expect only / separated names, and expect a leading / for absolute pathnames. They also expect multiple /'s to be treated as a single separator.

Even though NTFS supports case sensitivity for file names, the WIN32 API has no support for case sensitivity for directories and minimal support for case sensitivity for files, limited to a `FILE_FLAG_POSIX_SEMANTICS` creation flag for the `CreateFile()` function. Certain characters such as *, ?, >, |, :, ", and \, cannot be used in filenames created or accessed with the WIN32 API. The names, `aux`, `com1`, `com2`, `nul`, and filenames consisting of these names followed by any suffix, cannot be created or accessed in any directory through the WIN32 API.

Because Windows 95 doesn't support execute permission on files, it uses the `.exe` suffix to decide whether a file is an executable. Windows NT doesn't require this suffix, but some NT utilities, such as the DOS command interpreter, require the `.exe` suffix.

3.2 Line Delimiters

Windows NT uses the DOS convention of a two character sequence <cr><nl> to signify the end of each line in a text file. UNIX uses a single <nl> to signify end of line. The result is that file processing is more complex than it is with UNIX. There are separate modes for opening a file as text and binary with the Microsoft C library. Binary mode treats the file as a sequence of bytes. Text mode strips off each <cr> in front of each new-line as the file is read, and inserts a <cr> in front of each <nl> as the file is written. Because the number of characters read doesn't indicate the physical position of the underlying file, programs that keep track of characters read and use `lseek()` are likely to not work in text mode. Fortunately, many programs that run on Windows NT do not require the <cr> in front of each <nl> in order to work. This difference turned out to be less of a problem that we had originally expected.

3.3 Handles vs. file descriptors

The WIN32 API uses *handles* for almost all objects such as files, pipes, sockets, processes, and events, and most handles can be *duped* within a process or across process boundaries. Handles can be inherited from parent processes. Handles are analogous to file descriptors except that they are unordered, so that a per process table is needed to maintain the ordering.

Many handles, such as pipe, process, and event handles, have a synchronize attribute, and a process can wait for a change of state on any or all of an array of handles. Unfortunately, socket handles do not have this attribute. One of the few novel features of WIN32 is the ability to create a handle for a directory with the synchronize attribute. This handle changes state when any files under that directory change. This is how multiple views of a directory can be updated correctly in the presence of change.

3.4 Inconsistent Interfaces

The WIN32 API handle interface is often inconsistent. Failures from functions that return handles return either 0 or -1 depending on the function. The `CloseHandle()` function does not work with directory handles. The WIN32 API is also inconsistent with respect to calls that take pathname arguments and calls that take handles. Some functions require the pathname and others require the handle. In some instances, both calls exist, but they behave a little differently.

The WIN32 API is also inconsistent with respect to reporting errors when commands fail. Many commands return a boolean value for success or failure and the exit code for failure can be found by calling `GetLastError()`. However, a number of commands return the exit code with `0` indicating success.

3.5 Chop Sticks Only

The WIN32 subsystem does not have an equivalent for `fork()` or an equivalent for the `exec*()` family. There is a single primitive, named `CreateProcess()` that takes 10 arguments, yet still cannot perform the simple operation of overlaying the current process with a new program as `execve()` requires.

3.6 Parent/Child Relationships

The WIN32 subsystem does not support parent/child relationships between processes. The process that calls `CreateProcess()` can be thought of as the parent, but there is no way for a child to determine its parent. Most resources, such as files and processes, have handles that can be inherited by child processes and passed to unrelated processes. Any process can wait for another process to complete if it has an open handle to that process. There is a limited concept of process group that affects the distribution of keyboard signals, and a process can be placed in a new group at startup or can inherit the group of the parent process. There is no way to get or set the process group of an existing process.

3.7 Signals

The WIN32 API provides a structured mechanism for exception handling. Also, signals generated from within a process are supported by the API. However, signals generated by another process have no direct method of implementation. In addition to being able to interrupt processing at any point, a signal handler might perform a `longjmp` and never return.

3.8 Ids and Permissions

Windows NT uses *subject identifiers* to identify users and groups. A subject identifier consists of an array of numbers that identify the administrative authority and sub-authorities associated with a given user. A UNIX user or group id is a single number that uniquely identifies a user or group only within a single system. Information about users is kept in the a registry database which is accessible via the WIN32 API and the LAN manager API.

Windows NT uses an *access control list*, ACL, on each file or object to control the access of the file or object for each user. UNIX uses a set of permission bits associated with the three classes of users; the owner of the object, the group that the object belongs to, and everyone else. While it is possible to construct an access control list that more or less corresponds to a given UNIX permission, it is not always possible to represent a given access control list with UNIX permissions.

Windows NT has separate permissions for writing a file, deleting a file, and for changing the permission on a file. The write bit on UNIX systems determines all three. Thus, it is possible to encounter files that have partial write capability.

UNIX processes have real and effective user and group id's that control access to resources. Windows NT assigns each process a *security token* that defines the set of privileges that it has. UNIX systems use setuid/setgid to delegate privileges to processes. Windows NT uses a technique called *impersonation* to carry out commands on behalf of a given user. There is no user that has unlimited privileges as the *root* user does with UNIX. Instead the special privileges of root have been broken apart into separate privileges that can be given to one or more users. One of the biggest challenges we faced was providing the UNIX model of setuid/setgid on top of the WIN 32 interface.

The implementation of WIN32 for Windows 95 does not support the NT security model and calls return a *not implemented* error.

3.9 Terminal Interface

Windows NT and Windows 95 allow each character based application to be associated with a *console* which is similar to an `xterm` window. Consoles support echo and no echo mode, and line at a time or character at a time input mode, but lack many of the other features of the POSIX `termios` interface. There is no support for processing escape sequences that are sent to the console window. In echo mode, characters are echoed to the console when a read call is pending, not while they are typed. There are separate console handles for reading from the keyboard and writing to the screen.

3.10 Special Files

The WIN32 API supports unnamed pipes with the UNIX semantics. Named pipes are also supported but have different semantics than fifos and occupy a separate name space. There is no `/dev` directory to name special files such as `/dev/tty` and `/dev/null`. The WIN32 does support special names of the form `\\.\`*PhysicalDrive* for disk drives and tape drive devices.

Windows NT supports hard links to files, but there is no WIN32 API call to create these links. They do not support symbolic links in the file system directly, but on Windows 95 and on Windows NT 4.0, the file browser does support *short cuts* which are very similar to symbolic links.

3.11 Shared libraries

The WIN32 API supports the linking of shared libraries at program invocation and at run time. The libraries are called dynamically linked libraries or DLL's and are represented by two separate files. One file provides the interface and is needed at compile time to satisfy external references. The second file contains the implementation as is needed at run time.

There are some restrictions on DLL's that are not found on UNIX system shared library implementations. One restriction is that you cannot override a function called by a DLL by providing your own version of the function. Thus, supplying your own `malloc()` and `free()` functions will not override the calls to `malloc()` and `free()` made by other DLL's. Secondly, the library can only contain pointers to data, not data itself. Thus, making a symbol such as `errno` part of a DLL is impossible. Even making symbols such as `stdin` point to data in a DLL invites trouble since it is not possible to compile code that uses

```
static FILE *myfile = stdin;
```

3.12 Compilers and libraries

Microsoft sells the Visual C/C++ compiler for Windows NT and Windows 95. This compiler has both a graphical and command line interface. Microsoft also sells a software developers kit (SDK) that contains tools, including the Microsoft `nmake`. The compiler and linker use a different set of flags than standard UNIX compilers, and C files produce `.obj` files by default, rather than `.o` files. Fortunately, the linker can handle both `.obj` and `.o` files. The linker has options to choose a starting address and to specify whether the application is a console application, a GUI application, a POSIX application, or a dynamically linked library.

3.13 Environment Variables

The WIN32 API supports the creation and export of environment variables in much the same way that UNIX systems do. Some environment variables, such as `PATH` are used by both WIN32 and by UNIX, yet have different formats. UNIX uses a : separated list of pathnames; WIN32 uses a ; separated list.

4. UWIN DESIGN AND IMPLEMENTATION

We started work on writing our own POSIX library at the beginning of 1995 after being dissatisfied with the existing commercial products. We were able to put together a useful subset of functions in about 3 months. However, to be successful, it was necessary to provide as complete a package as possible. The library needed to handle console and serial line support, sockets, UNIX permissions, and other commonly used mechanisms such as memory mapping, IPC, and dynamic linking. In addition, to be useful, the libraries had to be documented and supported. This put the scope of the project outside of the reach of a small research department such as ours.

We subcontracted some of the development to Wipro in India to help complete this project. We jointly designed the terminal interface and the group in India implemented it. They also worked on completing the sockets library. They packaged the software for installation and are providing documentation. This section describes the UWIN implementation and how we solved many of the

problems described in Section 4.

4.1 UWIN Architecture

The current implementation of UWIN consists of two dynamically linked libraries named `posix.dll` and `ast52.dll` that more or less implement the functions documented respectively in section 2 and section 3 of UNIX manuals. In addition, a server process named UMS runs as `Administrator` (the closest thing to `root`). UMS generates security tokens for setuid/setgid programs as needed. It also is responsible for keeping the `/etc/passwd` and `/etc/group` files consistent with the registry database. The Architecture for UWIN is illustrated in Figure 1. The UMS server does not exist for Windows 95.

The `posix.dll` library maintains an open file table that is shared by all the currently active UNIX processes in a memory mapped region. This region is writable by all processes so that an ill-behaved process could affect another process. Even though all processes have read and write access to the shared segment, secure access to kernel objects in Windows NT is not compromised by this model because a process must have access rights to an object to use it; knowing its address or value doesn't give additional access rights. Some initial measurements indicated that the alternative of having a server process update the shared memory region, would have had a performance penalty that we did not believe was worth the cost. However, this is an area for future investigation.

The open file table is an array of structures of type `Pfd_t` as illustrated in Table 1.

Pfd_t	
long	refcount
int	oflag
char	type
short	extra

TABLE 1. File Table Structure

The `refcount` field is used to keep track of free entries in this table. The Win32 `InterlockedIncremenet()` and `InterlockedDecremenet()` functions are used to maintain this count so that concurrent access by different processes will work correctly. The `oflag` field stores the open flags for the file. The `type` field indicates what type of file, regular, pipe, socket, or special file. The function that is used read from or to write to the file depend on the value of `type`. For certain types, the `extra` field stores an index into a type-specific table that stores additional information about this file.

The `posix.dll` library also maintains a per process structure, `Pproc_t`. The per process structure contains information required by UNIX processes that is not required by Win32 processes such as parent process id, process group id, signal masks, and process state as illustrated in Table 2.

Like the open file table, the process table maintains a reference count so that process slots can be allocated without creating a critical region. The meaning of most of the fields in the process structure can be deduced by its name. The `Psig_t` structure contains the bit mask for ignored, blocked and pending signals. When the first child process is invoked by a process, a thread is created that waits for this and subsequent processes to complete. The `waitevent` field contains an event this thread also waits on so that additional children can be added to the list of children to wait for.

Pproc_t	
long	refcount
HANDLE	proc,thread
HANDLE	sigevent
HANDLE	waitevent
HANDLE	etok,rtok
ulong	ntpid
pid_t	pid,ppid,pgrp,sid
id_t	uid,gid
Psig_t	siginfo
mode_t	umask
ulong	alarmremain
int	flags
time_t	cutime,cstime
Pprocfd_t	fdtab[OPEN_MAX]

TABLE 2. Process Table Structure

The process structure contains an array of up to `OPEN_MAX` structures of type `Pprocfd_t` that is indexed by file descriptor. The `Pprocfd_t` structure contains the close-on-exec bit, the index of the file in the open file table, and the corresponding handle or handles as illustrated in Table 3.

The `posix.dll` library implements the `malloc()`, `realloc()`, and `free()` interface using the `Vmalloc` library written by Kiem-Phong

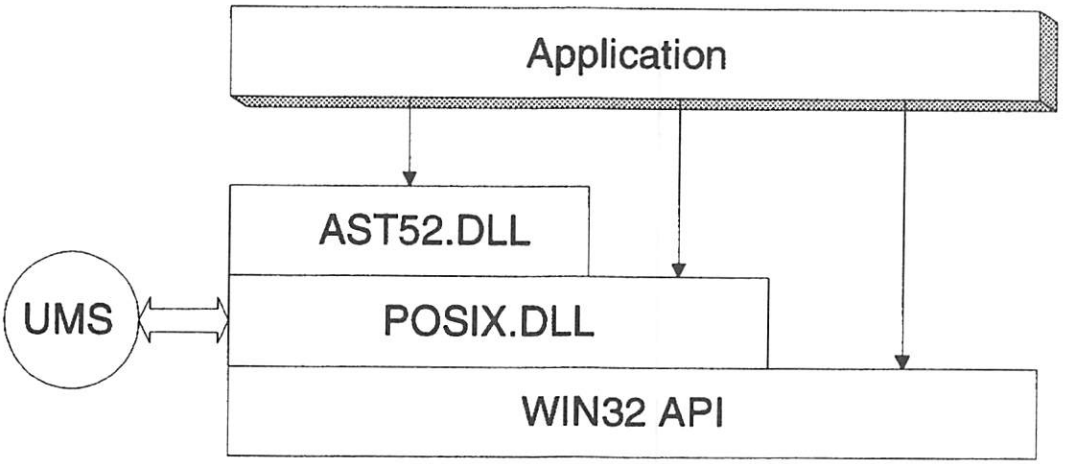


Figure 1. – UWIN Architecture

Vo[14]. The `Vmalloc` library provides an interface to walk over all memory segments that are allocated which is needed for the `fork()` implementation described later.

Pproc_t	
short	index
char	close_exec
HANDLE	primary
HANDLE	secondary

TABLE 3. Process file structure

The `asta52.dll` library provides a portable application programming interface that is used by all of our utilities. The interface to this library is named `libast.a`, for compatibility with its name on UNIX systems. The `52` is used to indicate that it is version 5.2 of this library, the latest version number on UNIX systems. `libast.a` provides C library functions that are not present on all systems so that application code doesn't require `#ifdefs` to handle system dependencies. `libast.a` is built using the `iffe` command [15] to feature test the host system and determine what interfaces do not exist in the native system.

`libast.a` relies on the Microsoft C library for much of the ANSI-C functionality. The most significant exception to this, other than `malloc()` which is provided by `posix.dll`, is the `stdio` library. `libast.a` provides its own version of the `stdio` library based on calls to `Sfio`[16]. The `Sfio` library makes calls to `posix.dll` rather than making direct calls to the WIN32 API as the Microsoft C library does so that pathnames are correctly mapped.

The use of `Sfio` also provides a simple solution to the <cr><nl> problem. When a file is explicitly opened for reading as a text file, an `Sfio` *discipline* for `read()` and `lseek()` can be inserted on the stream to change all <cr><nl> sequences are to <nl>. The `lseek()` discipline uses logical offsets so that the removal of <cr> characters is transparent. We did not provide a discipline to change <nl> to <cr><nl> since we discovered that most Windows 95 and Windows NT utilities worked without the <cr>s. The <cr>s could be inserted by a filter such as `sed` if required.

4.2 Files

The `posix.dll` library performs the mapping between handles and file descriptors. Usually, each file descriptor has one handle associated with it. In some cases, two handles may be associated with a file descriptor. An example of this is a console that is open for reading and writing which uses separate handles for reading and writing.

The `posix.dll` library handles the mapping between UNIX pathnames and WIN32 pathnames. Many UNIX programs assume that pathnames that

do not begin with a / are relative pathnames. In addition, only / is recognized as a delimiter. There is only a single root directory; the operation of changing to another drive does not change the root directory. The `posix.dll` library maps all file names it encounters. If the file name begins with a / and the first component is a single letter, then this letter is taken as the drive letter. Thus, the UNIX filename `/d/bin/date` gets translated to `d:\bin\date`. The file name mapping routine also recognizes special file names such as `/dev/tty` and `/dev/null`. A / not followed by a drive letter is mapped to the drive that UWIN has been installed on so that programs that embed absolute pathnames for files in `/bin`, `/tmp`, `/dev`, and `/etc` work without modification. The file name mapping also solves the problem caused by the WIN32 interface special treatment of the names, `aux`, `com1`, `com2`, `nul`, and filenames consisting of these names followed by any suffix,

Version 1.3 of UWIN adds support for Universal Naming Conversion (UNC) naming. UNC uses names of the form *//hostname/filename* to access files on a given host. In addition, Version 1.3 adds the name `/sys` as a way of naming the system directory. This makes it easier to write shell scripts that are portable across windows machines.

Finally, the path search algorithm was modified to look for `.exe` and `.bat` suffices.

One problem introduced by the pathname mapping is that passing file name arguments to native NT utilities is more difficult since it understands DOS style names, not UNIX names. A library routine was added to return a DOS name given a UNIX name.

The `posix.dll` library pathname mapping function also takes care of exact case matching on file systems that require it. One of the most troublesome aspects of the WIN32 API is its lack of support for pathname case distinction. It is not uncommon to have files named `Makefile` and `makefile` in the same directory in UNIX. UWIN handles case distinction by calling the WIN32 `CreateFile()` function both with and without the `FILE_FLAG_POSIX_SEMANTICS` function. If they compare equal, it executes the function internally, otherwise it spawns a POSIX subsystem process to carry out the task.

4.3 fork/exec

The `fork()` system call was implemented by creating a new process with the same startup information as the current process. Before executing `main()`, it copies the data and stack of the parent process into itself. Handles that were closed when the new process was created are duplicated into the new process. The `exec*()` family of functions was much harder to implement. The problem is that there is no way to overlay the calling process. Some commercial products have the current process wait for the child process to complete and then exit. There are two problems with this approach. First, a process that *exec*s repeatedly will fill up the process table. More importantly, resources from the parent process are not released. Our method causes the child process to be reparented to the grandparent and the process that calls `exec*()` to exit. The process id returned by the `getpid()` function will be the process id of the process that invoked the `exec*()` function. In other cases, it will be the same process id as the WIN32 uses. To prevent that process id from being used again by WIN32, a handle to the process is kept by the grandparent process.

Even though we implemented `fork()` and the `exec*()` family of functions, our code rarely uses them. Because the `CreateProcess()` function doesn't have an overlay flag, two processes need to be created in order to do both `fork()` and `exec*()`. `libast` provides a `spawn*()` family of functions that combines the functionality of `fork()/exec*()` on systems that don't have the `spawn*()` family. All functions in `libast` that create processes such as `system()` and `popen()` are programmed with this interface. On most UNIX systems, the `spawn*()` family is written using `fork()` or `vfork()` and `exec*()`. We implemented `spawn*()` in our `posix.dll` library to call `CreateProcess()` directly.

The `CreateProcess()` function has the ability to specify information and startup options that cannot be specified with `fork()` and `exec()`. The Version 1.3 of UWIN supplies a function `uwin_spawn()` that takes an argument that contains additional information for `CreateProcess()`. This should make it unnecessary for an application to call `CreateProcess()` directly.

4.4 Signals

Signals are handled by having each process run a thread that waits on an event. To send a signal to a process, the bit corresponding to the given signal number is set in the receiving process's process block, and then its signal thread event is set. The signal thread then wakes up and looks for signals. It is important for the signal handler to be executed in

the primary thread of the process, since the handler may contain a `longjmp()` out of the handler function. Prior to calling `main()`, an exception filter is added to the primary thread that checks for signals. The signal thread does this by suspending the primary thread raising an exception that will active the exception filter of the primary thread, and then resuming the primary thread.

4.5 Terminals

The POSIX `termios` interface is implemented by creating two threads; one for processing keyboard input events, and the other for processing output events and escape sequences. These threads are connected to the read and write file descriptors of the process by pipes. The same architecture is used for socket based terminals and serial I/O lines. Initially, these threads run in the process that created the console and make it the controlling terminal. These threads service all processes that share the controlling terminal. New threads will be created if the process that owns the threads terminates and another process is sharing the console. When a process is created, these threads are suspended and the console handles are passed down to the child. This enables a native application to run with its standard input and output as console handles. If the application has been linked with the `posix.dll`, then these threads are resumed before `main()` is called so that UNIX style terminal processing takes place. The result is that UNIX processes will echo characters as they are typed and respond to special keys specified by `stty`, whereas native WIN32 applications will only echo characters when they are read and will use Control-C as the interrupt character.

4.6 Ids and Permissions

Permissions for files are only available on Windows NT. Calls to get an set permissions return *not implemented* errors on Windows 95. Creating a Windows NT ACL that closely corresponds to UNIX permissions isn't very difficult. The ACL needs three entries; one for owner, one for group, and one that represents the group that contains all users. Windows NT allows separate permission to delete a file and to change its security attribute. These permissions are give to the owner of a file. The UNIX `umask()` command sets the default ACL so that native applications that are run by UWIN will create files with UNIX type permissions.

Mapping of subject identifiers to and from user and group ids is more complex. UWIN maintains a table of subject identifier prefixes, and constructs the user id and group id by a combination of the index in this table and the last component of the subject identifier. The number of subject identifier prefixes that are likely to be encountered on a given machine is much smaller than the number of accounts so that this table is easier to maintain.

4.7 Special files and Links

Special files such as fifos and symbolic links require `stat()` information that is not kept by the NT or FAT file systems. Also, the file system does not store the `setuid` and `setgid` permission bits. With the NT file system, this extra information has been stored by using a poorly documented feature called *multiple data streams* that allows a file to have multiple individually named parts. A separate data stream is created to hold additional information about the file. The `SYSTEM` attribute is put on any file or directory that has an additional data stream so that they can be identified quickly with minimal overhead during pathname mapping.

Using multiple data streams requires the NT file system. On other file systems, fifos and symbolic links are implemented by storing the information in the file itself. The setuid, setgid functionality is not supported on these file systems.

UWIN treats Windows 95 and Windows NT 4.0 *short cuts* as if they were symbolic links. However, these links can be created with any of the UWIN interfaces. This was done by reverse engineering the format of a short cut file and finding where the pathname of the file that it referred to was stored.

Fifos are implemented by using WIN32 named pipes. A name is selected based on the creation date of the fifo file. Only the first reader and the first writer on the fifo create and connect to the named pipe. All other instances duplicate the handle of either the reader or the writer. This way all writers to a fifo use the same handle as required by fifo semantics.

A POSIX subsystem command is also invoked to create hard links since there is no WIN32 API function to do this. Hard links fail for files in the FAT file system.

4.8 Sockets

Sockets are implemented as a layer on top of `WINSOCK`, the Microsoft API for BSD sockets. Most functions were straight forward to implement. The `select()` function proved more difficult than we had anticipated because socket handles could not be used for synchronization, and because the

Microsoft `select()` call only worked with socket handles. The `posix.dll select()` function allows different types of file descriptors to be waited for.

Our first implementation of `select()` created a separate thread that used the Microsoft `select()` to wait for socket handles, and created an event for the main thread to add to the list of handles to wait for. Our second implementation used a library routine to convert input/output events on sockets to windows messages and then waited for both windows messages and handle events simultaneously. This method had the added advantages that it was possible to implement `SIGIO` and that it was easy to add a pseudo file device named `/dev/windows` that could be used to listen for windows messages. Adding this pseudo device made it possible to use the UNIX implementation of `tcl` to port `tksh`[17] applications to Windows NT.

4.9 Invocation

When UWIN invokes a process, it does not know whether the process is a UWIN process or a native process. It modifies the `PATH` variable so that it uses the ; separated DOS format. It also passes open files in the same manner that the Microsoft C library does so that programs that are compiled with this library should correctly inherit open files from UWIN programs. The initialization function also sees whether a security token has been placed in its address space by the UMS server, and if so, it impersonates this token.

The POSIX library has an initialization routine that sets up file descriptors and assigns the controlling terminal starting the terminal emulation threads as required. The `posix.lib` library also supplies a `WinMain()` function that is called when the program begins. This function initializes the `stdin`, `stdout`, and `stderr` functions and then calls a `posix.dll` function passing the address of another `posix.lib` function that actually invokes `main()`. The `posix.dll` function starts up up the signal thread and sets the exception filter for signal processing as described above. The reason for this complexity is so that UNIX programs will start with the correct environment, and so that `argv[0]` will have UNIX syntax without the trailing `.exe` since many programs use `argv[0]`. Much of the complexity occurs inside the `posix.dll` part because programs do no require recompilation when changes are added there.

The current version of UWIN provides partial support for files larger than two gigabytes. The underlying NTFS file system supports 64 bit file offsets. However, the size of `off_t` is stored as a 32 bit integer because some programs would otherwise break. The type `off64_t` is defined and the functions `ftruncate64()`, `lseek64()`, and `truncate64()` have been implemented. The current `stat` structure actually fills in a 64 bit file size, but only the low 32 bits are accessible. The current version of `Sfio` supports 64 bit file offsets, but this option has not as of yet been enabled.

5. CURRENT STATUS

At the time of this writing, most interfaces required by the X/Open Release 4 standard have been written and work as described in the standard. The X/Open standard requires full ANSI C functionality as well. In addition, interfaces for the curses library, the sockets library, and the dynamic linking library, are also working.

A C/C++ compiler wrapper has been written that calls either the Microsoft Visual C/C++ 2.x, 4.x or 5.x compiler. This compiler supports the most commonly used UNIX conventions and implicitly sets default include files and libraries. In addition it has an added hook for specifying native compiler and linker options. Applications compiled with our `cc` command can be debugged with native debuggers such as the Visual C/C++ debugger. Several auto configuration programs use the output of the C preprocessor to probe the features of the system. The output format of the Microsoft C compiler caused some of the configuration programs to fail. To overcome this, a filter is inserted when running the compiler to generate preprocessor output so that existing configuration programs work. Our compiler wrapper can be invoked as `cc` for ANSI-C compilation, as CC for C++ compilation, and as `pcc` to build POSIX subsystem applications.

Our compiler wrapper follows the normal UNIX defaults for suffixes rather than using the Microsoft conventions; `.o`'s rather than `.obj`'s. The `.exe` suffix is not required for Windows NT since it uses permission bits to distinguish executables. However, since we also want binaries to run on Windows 95, the `.exe` suffix is added to the name of the output file if no suffix is supplied when the compiler is invoked as `cc` or CC.

The tools have been enhanced to make building dynamically linked libraries easier. The `ar` command has been extended with an option to

generate an export file from a definition file (`.def`). The `ld` command has a flag to build dynamically linked libraries.

The lastest version of `ksh`, `ksh-93` was ported. The implementation supports all features of `ksh-93` including job control and dynamic linking of built-in commands at run time. While no changes to the code should have been necessary, changes were made to `ksh` specifically for NT. The hostname mapping attribute, `typeset -H`, which has no effect on UNIX systems, was modified to call the `posix.dll` function that returns the WIN32 pathname corresponding to a given UNIX pathname. The ability to do case insensitive matching for file expansion was also added. A compile time option to allow <cr><nl> in place of <nl> was added to the shell grammar to avoid the overhead of text file processing.

About 175 UNIX tools have been ported to Windows NT, the vast majority required no changes. Common software development tools such as `yacc`, `lex`, `make` and `nmake` have also been ported. Most of the utilities are versions that we have written at AT&T over the last ten years and are easily portable to all UNIX platforms. Other utilities, such as `make`, `bc`, and `gzip` we compiled from the GNU source using `autoconfig` to generate headers and makefiles. The `yacc` and `less` utilities and the new `vi` program were ported from freely available BSD source code. In most cases, no changes were made to the original source code.

The X Windows code has two parts, the client and the server. The server had already been ported to Windows NT and Windows 95 by commercial vendors and there was no need to build UWIN version for it. In addition, the server is often running on a UNIX host. The most difficult part of porting the X Windows client code was the fact that it had `#ifdefs` for `WIN32` that selected native WIN32 calls, bypassing the UWIN calls. Once this was straightened out, the compilation was straightforward.

6. UNSOLVED PROBLEMS

It would be nice to say that all of UNIX could be implemented with WIN32, but this isn't the case. One problem, which is an artifact of using the the WIN32 API rather than the POSIX subsystem, is that there is no way to create or access a file whose name ends in '.', even by using the `FILE_FLAG_POSIX_SEMANTICS` flag with `CreateFile()`.

A second problem is that the way authentication works in Windows NT differs from that on UNIX systems. On UNIX systems, the password is encrypted and compared to the encrypted password in the user accounts database. On Windows NT, a function that takes the user name and password is called, and this function returns a *token* the can be used to defined the access privileges of a process. Since there was no access to the encrypted passwords with WIN32, we had to make changes to programs such as `ftp` that require authentication or programs such as `telnetd` that need to create processes on behalf of the user.

A third problem for which we were not able to find a satisfactory solution was how to `fchmod()` a file whose handle was opened with read permission only. If the handle was opened by another process the name is not known, so a separate open is not possible. One solution would be to always try opening the file with permission to change the mode. The problem with this solution is that the open will fail for any file that is not owned by the user opening the file, and a second open attempt would be required. This would practically double to time needed to open a file for opening a readonly only, which we felt was unacceptable.

We encountered a number of problems relating to concurrency restrictions that do not occur on UNIX systems. For example, if a file was memory mapped, an attempt to open it for truncation would fail. We could not find a way around this problem so we had to change the `vi` code that we were using to not use memory mapping. We are unable to change the access permissions of a dynamically linked library that is in use. We are also unable to rename a directory any another process has this directory, or a subdirectory as its current working directory. In Windows 95, we are unable to move or rename files that are open, even when these have been opened for maximum sharing.

Finally, we encountered numerous problems with the permission system. We were unable to add a new group to an existing process. It is often difficult or impossible to map access control lists to UNIX permissions in a meaningful way.

7. FUTURE WORK

We currently are using the Microsoft Developers Studio debugger. Unfortunately, this requires us to be logged in on the console. We often have several users logged onto a single NT workstation and only one user at a time can use this debugger. In

addition, the debugger can't be used when logged in from home via a telnet session. We hope to write the nub for the `deet` debugger[18] so that it can be used as a command line and/or graphical debugger.

In addition, we are trying to decide how to port the *n* dimensional file system, *n*-DFS[19], to Windows NT. *n*-DFS provides a mechanism to add file system services such as viewpathing and versioning. The difficulty in porting *n*-DFS is that it must also capture native WIN32 API calls to provide a transparent interface.

We would like to add support for asynchronous I/O as defined by the POSIX realtime standard[20]. Unfortunately, unlike the POSIX API, the WIN32 API requires that you make the decision about whether the I/O is synchronous or asynchronous when the file is opened as well as when it is read. A file opened for synchronous reads cannot be read asynchronously, and a file opened for asynchronous reads cannot be read synchronously. This makes it impossible to use asynchronous I/O on a file that has been redirected by the shell.

The current version of UWIN does not handle many of the internationalization issues well. The current implementation has been compiled for ASCII rather than UNICODE. We plan to use UFT8 encoding of UNICODE for the system call interface, and to convert to UNICODE on the NT file system. This way we do not need to build separate binaries for UNICODE.

Another issue worth investigating is whether it is possible to run Linux binaries under UWIN. This would only make sense for dynamically linked programs.

Finally there are some WIN32 interfaces that could be handled through the file system interface such as the Windows NT registry and the clipboard.

8. CONCLUSIONS

There appear to be few if any technical reasons to move from UNIX to Windows NT. The performance of Linux exceeds that of NT 4.0 and Linux appears to be more reliable. However, if you want to or need to move an application to Windows 95 or Windows NT, we believe the POSIX library we developed to be superior to any of the existing commercial libraries.

The code for the `posix.dll` library is fairly small, about 15K lines including the terminal emulator. This library runs in the WIN32 subsystem using the WIN32 API and runs under Windows 95 as well.

The UWIN binaries are freely available for non-commercial use from the web site `http://www.research.att.com/sw/tools/uwin`. We hope that this will encourage contributions of applications that have been built with UWIN. Licenses for commercial use of UWIN are available from Global Technologies, Ltd., `http://www.gtlinc.com`.

REFERENCES

1. *Microsoft Win32 Programmer's Reference, Volume 2* Microsoft Press, 1993.
2. Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books, 1995.
3. *The X/Open Release 4 CAE Specification, System Interfaces and Headers*, Issue 4, Vol. 2, X/Open Co., Ltd., 1994.
4. David Korn, *Porting UNIX to Windows NT*, Proceedings of the Anaheim Usenix, pp. 43-58, 1997.
5. *Practical Reusable UNIX Software*, Edited by Balanchander Krishnamurthy, John Wiley & Sons, 1995.
6. *The OpenEdition MVS Users Guide,IBM,* 1996.
7. M. Accetta et al., *Mach: A New Kernel Foundation for Unix Development,* Usenix Association Proceedings, Summer 1986.
8. Jeffrey Richter, *Advanced Windows – The Developers Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, 1995.
9. *POSIX – Part 1: System Application Program Interface,* IEEE Std 1003.1-1990, ISO/IEC 9945-1,1990.
10. Richard A. Becker, John M. Chambers, and Alan R. Wilks, *The New S Language : A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole, New Jersey, 1988.
11. Morris Bolsky and David Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.
12. Glenn S. Fowler, *A Case for Make,* Software – Practice and Experience, Vol. 20, No. S1, pp. 30-46, 1990.
13. M. McKusik, W. Joy, S. Leffler, and R. Farbry, *A Fast File System for UNIX*, ACM Transactions on Computer Systems, Vol. 2, No. 3, August, 1984, 181-197.
14. Kiem-Phong Vo, *Vmalloc - A General and Efficient Memory Allocator*, Software – Practice and Experience, Vol. 26, No. 3, pp 357-374, March 1996.
15. Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo, *Feature Based Portability*, Proceedings of the USENIX Symposium on Very High Level Languages, 1994.
16. David Korn and Kiem-Phong Vo, *SFIO - A Safe/Fast String/File I/O,* Proceedings of the Summer Usenix, pp. 235-255, 1991.
17. Jeffrey Korn, `Tksh`*: A Tcl Library for KornShell*, Fourth Annual Tcl/Tk Workshop, Monterey, CA, July 1996, pp 149-159.
18. David R. Hanson and Jeffrey L. Korn, *A Simple and Extensible Graphical Debugger*, Proceedings of the Anaheim Usenix, pp. 173-184, 1997.
19. Glenn Fowler, David Korn and Herman Rao, "*n*-DFS The Multiple Dimensional File System", Trends in Software – Configuration Management, pp. 135-154, 1994.
20. *POSIX – Part 1: System Application Program Interface, Amendment 1: RealTime Extension* IEEE Std 1003.1b-1993 ,1993.

Implementing Security and Mobility Functions in Kernel Drivers

Yoshiyuki Tsuda, Masahiro Ishiyama, Atsushi Fukumoto and Atsushi Inoue
R&D Center, Toshiba Corporation
Ken-ichi Yokoyama
Fuchu Works, Toshiba Corporation

Abstract

We are developing a secure, mobile network system, named "Network CryptoGate (NCG)", which provides a secure Virtual Private Network (VPN) environment for mobile users seamlessly, even if they move to an insecure network. NCG is designed upon IETF standards, that is IP security (IPSEC) and Mobile IP (MobileIP), in order to make the whole system interoperable with other implementations. Currently, we have developed an NCG client software on Windows NT [1]. At the poster/demonstration session, we will demonstrate how an NCG client works.

1. System Overview

There are two major components in the NCG network system: NCG servers and NCG clients, as illustrated in Figure 1.

As a VPN server, an NCG server performs encryption and authentication for all packets that leave the private network, while for all incoming packets it decrypts and checks the authentication. As a mobility agent, an NCG server intercepts those packets which bound for a moved NCG client, and transfers them to the NCG client's current location. We have already implemented NCG servers are on BSD and Solaris [2], and are currently porting them to Windows NT [1].

An NCG client is a client software on a mobile terminal. When a mobile terminal leaves the private network, an NCG client encrypts and authenticates all packets bound for the private network, and decrypts all packets received from that network and checks their authentication. Also, an NCG client performs Mobile IP functions as a mobile terminal. We have developed NCG clients on Windows NT, and are now porting them to Windows 95 [1].

[1] "Windows NT" and "Windows 95" are registered trademarks of *Microsoft Corporation.*

[2] "Solaris" is a registered trademark of *Sun Microsystems, Inc..*

2. NCG client software architecture

A current NCG clients consists of an NDIS driver, a transport driver and application program, as illustrated in Figure 2. When a mobile terminal leaves the private network, all packets from/to the TCP/IP driver are processed by the MobileIP/IPSEC modules in the transport driver so as to preclude any impact on the TCP/IP driver and the upper software modules, as well as the card driver. Thus, we can provide a secure, seamless VPN environment for mobile users.

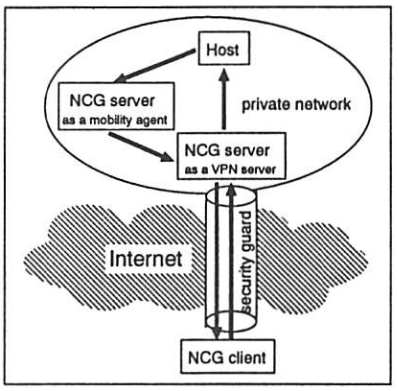


Figure 1. NCG system overview

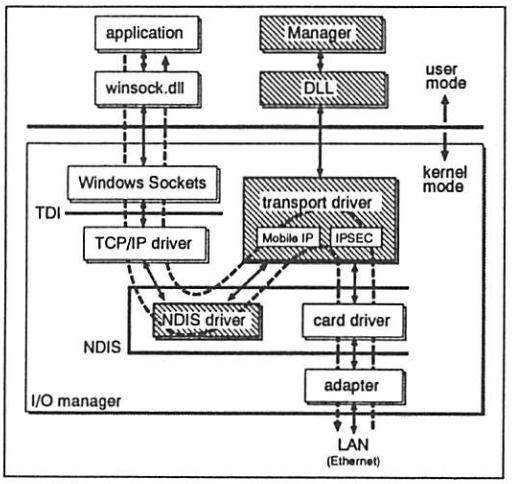


Figure 2. NCG client software architecture

Millipede: a User-Level NT–Based Distributed Shared Memory System with Thread Migration and Dynamic Run-Time Optimization of Memory References

http://www.cs.technion.ac.il/Labs/Millipede

Ayal Itzkovitz Assaf Schuster Lea Shalev
Computer Science Department, Technion, Haifa

Abstract

MILLIPEDE is an all user mode, no kernel-patches, "add on" software tool for standard corporate environments, that takes advantage of idle system resources and efficiently utilizes idle processor time in available distributed environments of personal workstations. MILLIPEDE presents to the user a powerful virtual parallel machine which abstracts away the underlying hardware configuration. In this way MILLIPEDE supports mapping of the applications to dynamically varying levels of parallelism according to both changes in the underlying hardware and changes in the application requirements.

MILLIPEDE is multi-threaded, thus taking full advantage of SMPs. MILLIPEDE provides a true distributed shared memory with several coherence protocols (and a flexible mechanism for easy inclusion of new ones) and dynamic thread migration [3]. MILLIPEDE studies the memory access pattern and optimizes the locality of memory references by adapting the thread distribution accordingly [5].

MILLIPEDE supports several of the more liberal parallel programming paradigms [2]. Currently we support PARC (which allows for example nested parallelism and barriers among sibling activities), PARC++ (which is as flexible as C++ except for additional parallelizing constructs), the SPLASH macros (which we had to adapt to Windows-NT and to the multi-threaded concept), and of course for best speedups one can use directly the MILLIPEDE job manager library. We are working on the implementation of PARFORTRAN90 and JAVA (for which we came up with a new definition of the JVM memory behavior, and with an efficient algorithm for distributed garbage collection [4]).

In order to support many different programming languages on top of a single virtual parallel machine we developed MILLIPEDE Job Event Control (MJEC) [2]. MJEC can be used to efficiently implement a variety of synchronization and communication protocols (e.g., PARC in about 250 lines of code).

MILLIPEDE is fully implemented at the Technion, Haifa, using the Windows-NT operating system (previous version on MACH [1]). We refer the reader to our WWW site (see above) for online versions of the papers, and for downloading a distribution of MILLIPEDE.

References

[1] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice & Experience*, 1997. (To Appear). Also Technion/LPCR/TR, #9506, November 1995.

[2] A. Itzkovitz, A. Schuster, and L. Shalev. Millipede: Supporting Multiple Programming Paradigms on Top of a Single Virtual Parallel Machine. In *Proc. HIPS Workshop*, Geneve, April 1997.

[3] A. Itzkovitz, A. Schuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *The Journal of Systems and Software*, 1997. To appear (See also Technion TR LPCR-#9603).

[4] D. Kogan and A. Schuster. Collecting Garbage Pages with Reduced Memory and Communication Overhead. In *Proc. European Symposium on Algorithms*, Graz, September 1997.

[5] A. Schuster and L. Shalev. Access Histories: How to Use the Principle of Locality in Distributed Shared Memory Systems. Technical Report #9701, Technion/LPCR, Jan 1997. Submitted for Publication.

High Performance Web Servers on Windows NT
Design and Performance*

James C. Hu, Irfan Pyarali, and Douglas C. Schmidt
Washington University in St. Louis

Abstract

This research provides two contributions to the study of high-performance Web servers. First, it outlines the optimizations necessary to build efficient and scalable Web servers and illustrates how we applied some of these optimizations to create JAWS, a high-performance Web server that is explicitly designed to alleviate overheads incurred by existing Web servers on high-speed networks. Second, this paper describes how we have customized JAWS to leverage advanced features of Windows NT, such as asynchronous mechanisms for connection establishment and data transfer. Our work includes performance results which characterize the effectiveness of these techniques under increasing server load conditions. We conclude that optimal performance requires adaptive server behavior.

1 JAWS Overview

JAWS is the Web server prototype we developed to analyze Web server performance bottlenecks. Our research involves the empirical study and analysis of the impacts different optimization strategies have to Web server performance as the Web server is subjected to various load conditions, such as request hit rate and requested file size. The strategies under study include: *I/O Strategies*, such as Asynchronous, Synchronous, and Reactive I/O; *Caching Strategies*, such as LRU, LFU, Hinted, and Structured; *Concurrency Strategies*, such as single threaded, thread-per-request, thread-per-session (persistent connections), and thread pool; *Request Handling Strategies*, such as prioritized requests, parallelized protocol processing, and content negotiation; and *Adaptive Protocols*, such as protocol negotiation (PEP), and dynamic protocol pipelines.

2 Performance Results

As shown in [1], JAWS consistently outperforms the other servers in our test suite. These servers included Apache, PHTTPD, Roxen, Netscape Enterprise Server, Zeus, and W^{3}C Jigsaw. During the study, we analyzed the results of our experiments to discover key Web server bottlenecks. We identified the following two key determinants of Web server performance: concurrency and event dispatching strategies; and filesystem access.

Additional experiments described in [2] characterize the relative impacts of different I/O models coupled with different concurrency strategies under various loads on Windows NT over a 155 Mbps ATM network. These experiments revealed two important results. First, throughput is highly sensitive to the I/O strategy and file size. Synchronous I/O mechanisms were found more appropriate for smaller files, while the asynchronous `TransmitFile` appeared most effective for larger files. Second, latency is highly sensitive to hit rate. For smaller files, synchronous I/O provided consistently better performance. For larger files under light loads, `TransmitFile` provided better latency, but is significantly worse than synchronous I/O under heavier loads.

3 Conclusions

These results illustrate that no single Web server configuration is optimal for all circumstances. In order to achieve optimal performance, a Web server must be designed to utilize both *static* adaptivity (bindings of common operations to high performance mechanisms of the native OS) and *dynamic* adaptivity (altering run-time behavior "on-the-fly" based on present load conditions). JAWS provides an application framework which makes this possible on Windows NT.

References

[1] James Hu, Sumedh Mungee, and Douglas C. Schmidt. Principles for Developing and Measuring High-performance Web Servers over ATM. In *Submitted for publication (Washington University Technical Report #WUCS-97-10)*, February 1997.

[2] James Hu, Irfan Pyrarli, and Douglas C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Submitted to the 2nd Global Internet Conference*. IEEE, November 1997.

*Additional information on this research is available from the website `http://www.cs.wustl.edu/~jxh/research/`.

IntelliJuke - a Caching Jukebox-Based Storage Server

Yitzhak Birk, Uri Kareev and Mark Mokryn
Technion - Israel Institute of Technology
Haifa 32000, Israel
birk@ee, kareev,mark@psl.technion.ac.il; www.psl.technion.ac.il

Background

The cost-effectiveness of CDs applies only to the storage medium itself; once a drive is included, CD-ROM often becomes inferior to magnetic hard drives. Moreover, a CD is often far from full, thus further reducing its cost-effectiveness. (For this reason, DVD will not help in many cases.)

Jukeboxes (AKA changers) feature high volumetric storage density with moderate communication bandwidth and multi-second access times. Jukebox manufacturers have focused on building very robust systems with high-performance robotics in order to mitigate the intrinsic performance shortcomings, resulting in a price-per-slot of $30-$120 in late 1996!

We believe that jukeboxes should provide access to a very large number of CDs at minimal cost, and that caching on magnetic disk drives should provide the performance. A byproduct of caching is reduced jukebox activity, so the jukebox's robustness may be reduced without increasing the mean time between failures. Our focus is on efficient caching.

Caching approach

We have adopted a hybrid caching approach: speculative fetching based on "intelligence", typically involving entire files and even related files, with removal based on "de facto" information and applied at a much finer, intra-file granularity. Our rationale is as follows: the miss penalty is very high; the cache (disk) is relatively inexpensive and can be large, possibly permitting an unaccessed item to reside in the cache for several days before it must be removed. If, however, in the course of several days in the cache, certain blocks in a given file were accessed while others were not, there is reason to believe that the unaccessed blocks will not be accessed in the near future and can be discarded. It should be noted that this observation hinges not merely on the relative access times of the two storage layers; rather, it takes into account the time constants of a human user!

Since it is easy to construct "good" and "bad" scenarios for such an approach, we decided to build a prototype so as to permit experimentation and measurements in a real environment.

IntelliJuke overview

IntelliJuke is a network-accessible hierarchical storage server comprising a PC running Windows NT, connected to a Kubik CDR-240 juke box with 240 slots and two 12X Plextor CD-ROM drives. Data is presented to users as a collection of NTFS directory trees on a hard drive, one tree per CD (single drive letter for all). The initial goal is to enable users to seamlessly access any data that would be accessible if the entire CD were copied onto a hard drive using file manager. Our focus has been on providing support for novel caching policies while using the native NT services whenever possible. The project is presently in advanced implementation stages at the Parallel Systems Lab of the EE dept., with most of the difficult problems out of the way. The caching algorithms will be tuned once the system is fully operational.

NT- related implementation challenges

- The hybrid caching approach and requirement for network access mandated the insertion of a "highest level" driver above an NTFS partition.
- Construction of a communication mechanism between our kernel driver and user-mode code. One complication is that communication is initiated by the driver.
- Overcoming the locking of files by the operating system. This occurs, for example, when the OS decides to issue a read-ahead request while we are handling an earlier miss to the same file, and prevents us from writing the requested data into the cache.
- The desire to manage the cache at block granularity and to free up disk space while using NTFS forced us to support "sparse" NTFS files.

Acknowledgments and credits. Parts of IntelliJuke were implemented by Amnon I. Govrin, Ran Herzberg, Eran Rosenberg and Eyal Zangi. Tomer Kol and Evgeny Rivkin have provided ongoing assistance. The project has been supported in part by Microsoft through product donations and by a grant from EMC (Israel) Storage Systems Ltd.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems.

USENIX and its members are dedicated to:
- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's bi-monthly newsletter featuring technical articles, system administration tips and techniques, SAGE News, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as nine technical meetings every year.
- Discounts on the purchase of proceedings from USENIX conferences and symposia and other technical publications.
- Discount on purchases of USENIX CD-ROMs.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates and Prentice Hall PTR.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Special subscription rate for *The Linux Journal.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Andrew Consortium
Apunix Computer Services
Boeing Commercial
Crosswind Technologies, Inc.
Earthlink Network, Inc.
ISG Technologies, Inc.
Matsushita Electric Industrial Co., Ltd.
Motorola Research & Development
MTI Technology Corporation
O'Reilly & Associates
Sybase, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Bluestone, Inc.
Enterprise Systems Management Corp.
Great Circle Associates
OnLine Staffing
Paranet, Inc.
Pencom Systems Administration/PSA
Southwestern Bell
Taos Mountain
Texas Instruments, Inc.
TransQuest Technologies, Inc.

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-88-X